

AD-A087 412

YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE

F/6 9/2

DRAFT SOFTWARE METRICS PANELS FINAL REPORT. PAPERS PRESENTED AT--ETC(U)

JUN 80 A J PERLIS, F G SAYWARD, M SHAW

N00014-79-C-0672

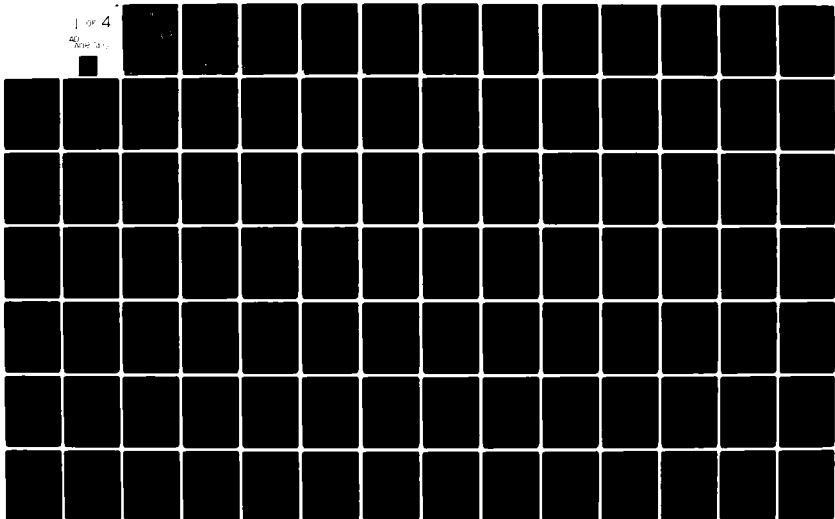
UNCLASSIFIED

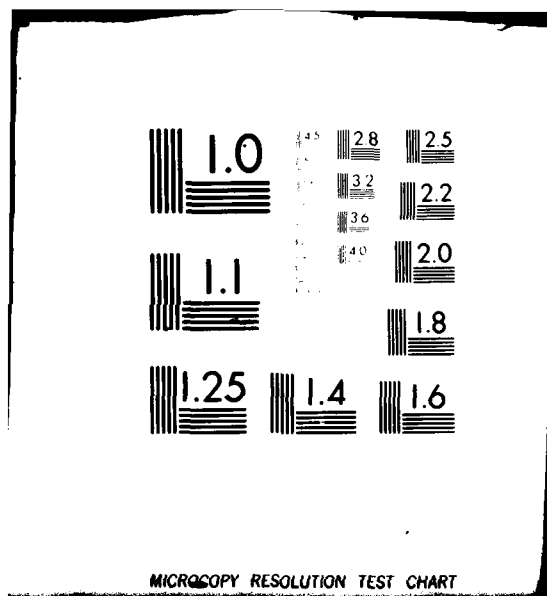
RR-182/80

NL

1 OF 4

40
White Card





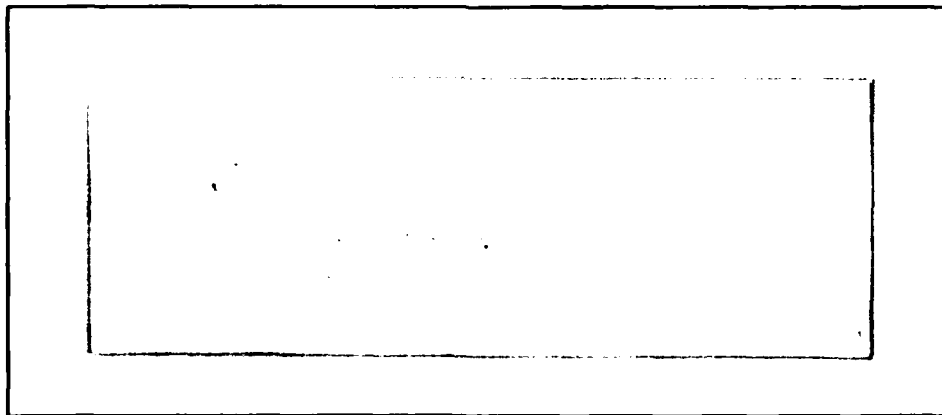
LEVEL ~~11~~

(12)
64

ADA 087412



DTIC
ELECTE
JUL 30 1980
S D C



This document has been approved
for public release and sale; its
distribution is unlimited.

DDC FILE COPY

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

80 7 7 046

12

DTIC
ELECTE
JUL 30 1980

12-5911

11/Jul 80

Draft Software Metrics Panel Final Report
Papers Presented at the 30 June 1980
Meeting on Software Metrics, Washington DC.

10 Alan J. Perlis, Frederick G. Sayward
and Mary Shaw, editors

9 Research Report #182/80
June 1980

14 RR-182/80

15

This research was supported in part by the Office of Naval Research
under Research Contract N00014-79-C-0672

All the material contained in this document is in draft form. As
a courtesy to the authors,

This document has been approved
for public release and sale; its
distribution is unlimited.

4107051

Jon

Contents

Preface	1
Introduction	1-1
The Life Cycle Model	
The Role of Software Metrics in Life Cycle Decisions	
Observations and Limitations of the Study	
The State of the Art and Research Recommendations	
Toward a Scientific Basis for Software Evaluation	2-1
J. Browne - University of Texas	
M. Shaw - Carnegie-Mellon University	
Design of Software Experiments	3-1
F. Sayward - Yale University	
Data Collection, Analysis and Validation	4-1
V. Basili - University of Maryland	
Experimental Evaluation of Software Characteristics and Programmer Performance	5-1
B. Curtis - General Electric Company	
Software Forecasting	6-1
R. DeMillo - Georgia Institute of Technology	
R. Lipton - Princeton University	
Controlling Software Development Through The Life Cycle Model	7-1
A. Perlis - Yale University	
Resource Models	8-1
V. Basili - University of Maryland	
High Level Language Metrics	9-1
J. Sammet - IBM Corporation	
Performance Evaluation: A Software Metrics Success Story	10-1
J. Browne - University of Texas	
W. Lynch - Xerox Corporation	
Statistical Measures of Software Reliability	11-1
R. DeMillo - Georgia Institute of Technology	
F. Sayward - Yale University	
The Measurement of Software Quality and Complexity	12-1
B. Curtis - General Electric Company	
Complexity of Large Systems	13-1
L. Belady - IBM Corporation	

Software Maintenance Tools and Statistics in the Life Cycle of a Computing Application 14-1
M.Muller - World Bank

A Scientific Approach to Statistical Software 15-1
I. Francis - Cornell University

When is "Good" Enough? Evaluating and Selecting Software Metrics 16-1
M. Shaw - Carnegie-Mellon University

Annotated Bibliography

Accession For	
NTIS GMA&I	<input checked="" type="checkbox"/>
DDC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<i>Per 182</i>
By	<i>[Signature]</i>
Distribution/	
Availability Codes	
Dist	Available/or Special
<i>A</i>	

RE: Processing of Draft Copy
Draft copy should be processed per Dr. Robt.
3. Grafton, ONR/Code 437

PREFACE

During the Spring of 1979 Marvin Denikoff of the Office of Naval Research asked for our opinion on the merit of the research which was being conducted on "software metrics." After a quick review of the literature it became apparent that the central issues revolved around the following question:

Can there be assigned to software and the processes associated with its design, development, use, maintenance, and evolution indices of merit that can support quantitative comparisons and evaluation of software?

We decided that an in-depth study was needed to identify useful areas in software metrics and software experimentation, to analyze the research being done in these areas, and to recommend directions which these areas should follow in the near term. Eventually this work would lead to the development of a scientific basis for analyzing and evaluating software which is so desperately needed by those involved with software management.

We recognized early that a large fraction of the computer science research community held little hope for software metrics. Consequently we decided that part of our effort should be in focusing the attention of the research community on the problems and methods in software metrics.

We perceived that these ends could best be met by organizing two groups of scientists: an Advisory Group and a Study Panel. Each member of the Study Panel would be assigned a software metric area to study and evaluate. The Advisory Group would be composed of distinguished senior scientists who would provide on-line advice to the Study Panel. The members selected for the two groups were

Advisory Group

Allen Newell, Carnegie-Mellon University
Jack Schwartz, Courant Institute

Study Panel

Vic Basili, University of Maryland
Les Belady, IBM Corporation
Jim Browne, University of Texas
Bill Curtis, General Electric Company
Rich DeMillo, Georgia Institute of Technology
Ivor Francis, Cornell University
Richard Lipton, Princeton University
Bill Lynch, Xerox Corporation
Merv Muller, World Bank
Alan Perlis (Chairman), Yale University
Jean Sammet, IBM Corporation
Fred Sayward (Assistant Chairman), Yale University
Mary Shaw (Assistant Chairman), Carnegie-Mellon University

An organizational meeting was held at Yale University September 10-12, 1979 at which time an initial state of the art presentation was given and the software metrics issues were discussed. Members were initially asked to study the current status of a variety of software metric areas.

A second meeting was held on January 31-February 1, 1980 immediately following the Principles of Programming Languages Conference at Las Vegas. State of the art evaluations were presented and final topics were assigned.

The draft papers in this report contain state of the art evaluations and directional recommendations for several areas of software metrics. They are to be presented at an open meeting to be held in Washington, DC on June 30, 1980. Efforts have been made to have a cross section of the computer science research community attend this meeting.

Our plans call for taking the final versions of the papers contained in this report and the questions, answers, and comments made at the June 30 meeting and integrating them into a book to be published during the Fall of 1980. We hope that this volume will serve as a focal point in the development of a science of software metrics.


Alan Perlis
Fred Sayward
Mary Shaw

INTRODUCTION

The Life Cycle Model

Software metrics is a new area of computer science aimed at assigning quantitative indices of merit to software. Here software means more than simply source code; ~~we use~~ 'software' as a generic for all the stages of tailoring a computer system to solve a problem, *IS USED HERE*. All software passes through the following seven stages in its life cycle:

- (1) requirements analysis,
- (2) specification,
- (3) design,
- (4) implementation,
- (5) testing and integration,
- (6) maintenance and enhancement, *AND*
- (7) replacement or retirement.

Since software specification is always imprecise and since the demands on software change with time, backtrack cycles to earlier stages often take place. It is not uncommon for several stages to co-exist and influence each others' progress. 

The Role of Software Metrics in Life Cycle Decisions

The purpose of software metrics is to help answer the questions which arise as the life cycle progresses. Two questions common to all the stages are:

Is it time to go onto the next stage?

Is a backtrack to an earlier stage needed?

At each stage there is also a set of questions about the software and the project whose precise answers are needed to dictate an optimal life cycle control flow. Software metrics addresses such questions but it is not sufficiently developed yet to provide precise answers to most of them.

1) Requirements Analysis

While this area of system development has an enormous influence on system software, the questions of concern here must lie outside the domain of software metrics -- for it is not until the requirements are fixed that the structure of the software can begin to take shape. What is needed is a link from requirements analysis to software specification, but this link must necessarily not be software-oriented, just as a link from an informal to a formal model cannot be formal.

2) Specification

At this stage an informal statement of the problem and its proposed solution has been prepared. Questions to be answered include:

What is the cost of production?

What are the memory requirements of the software?

What are the speed requirements of the software?

How long will it take to produce?

When will it have to be replaced?

What manpower loading should be used?

Is the project feasible? That is, does the expected production time exceed the time when the software will be of use?

3) Design

At this state a detailed formal statement of the problem and its proposed solution have been prepared. This includes a development plan for all future stages of the life cycle. Questions to be answered include:

What machine configuration to use?

What language to use?

Is it possible to incorporate the work of others or must everything be built in house?

How will the availability of tool X affect factor Y?

How close to its limits is the system expected to run?

What are the potential future enhancements?

Should the system be all encompassing from which subsystems are carved out or should the system be primitive on which specific systems are built?

4) Implementation

Some questions need to be answered before implementation begins and others arise during implementation. They include:

What developmental technology should be used? Should the system be built all at once or should it be constructed through a sequence of executable prototypes?

What programming discipline should be used? Chief programmer? Cottage industry?

Is the project on schedule?

Is the project on the budget?

Is the implemented code correct? If not, how close is it to meeting the specification?

What is the quality of the implemented code? Is it understandable? Is it maintainable? Is it enhanceable?

5) Testing and Integration

At this stage the chief question is: Does the implementation meet the specification? This usually reduces to questions concerning what the implementation actually does, what resources it uses, and how easy it is to use. Such questions can be asked about individual modules and about integrated modules working in concert. The decisions to be made include:

Should testing be done top down or bottom up?

Which of the available testing methodologies should be used?

What levels of satisfactory testing are sufficient?

How well does the testing environment approximate the execution environment?

How will subsequent error reports be handled?

6) Maintenance and Enhancement

These two very different activities are often linked because both result in a re-release of the system. Maintenance is similar to testing except that the software execution environment has changed from a controlled world of testing to the hurly-burly of actual use. Every repair and update must be tested, so the questions generated by maintenance are similar to the ones above in the testing stage. Enhancement, on the other hand, is a post-release augmentation of the system specifications to meet unforeseen demands. Enhancement may cause a backtrack all the way to the requirements analysis stage. The questions we ask about enhancement include:

What is the cost of the enhancement? Is it worthwhile?

Will the enhancement speed up or delay replacement?

What is the re-release strategy?

Once it has been decided that an enhancement should take place, there is an automatic feedback at least to the specification stage of the life cycle.

7) Replacement and Retirement

Among the questions asked when considering replacement or retirement of a system are:

Has the problem outgrown the program?

Has technology moved beyond the program?

Has a critical support resource for the system become unavailable?

Would it cost less to re-build the system than to maintain and enhance the system?

How should the system be phased out?

Should there be a change in the language in which the system is written? In the machine on which the system runs?

Observations and Limitations of the Study

Because of the time constraints on our study, the size of the problems under investigations, and the extremely uneven level of quality attained by software projects, we have not attempted to present a complete survey. We have concentrated on those issues in software metrics that require and will benefit from near term research.

Except in a small way we have not addressed the issue of manpower, although undoubtedly the competence, availability, and training of manpower plays an important role in defining the set of metric-oriented questions for a particular software project.

We have come to recognize the existence of certain critical issues in relating software metrics to other areas of computer science. For example, software should be seen as very different from algorithms -- not so much in representation but in size, variability, and rate of change. In software, evolutionary complexity is probably more important than the classical time and space measures with which computer science has been concerned so far.

There is a tendency in studying software (or anything else, for that matter) to be satisfied with mere curve-fitting rather than with developing models whose inherent structure illustrates one or more important aspects of the topic being investigated. Indeed, throughout our work we have had a growing awareness that the present software management needs overwhelm current insights in computer science. Needs have overwhelmed insights to such an extent, and there has been so little order, discipline, and content in the resulting practice of software metrics, that a significant portion of the computer science research community is completely turned off and chooses not to perform research in software metrics. But it is important that computer science not focus on premature or incorrect developments to discredit the field. Software will grow more complex, not simpler, and the need for software metrics will grow apace.

No matter what aspect of software one studies, there is a noticeable lack of collected and categorized field data on which to build. RADC has recently initiated the categorization of the information which does exist, but since past software projects have rarely integrated data collecting into their production schedule there is really not much to go on. Serious large-scale data collection is imperative if order is to be brought into the software field.

Of course, collected field data is more meaningful when related to the data and conclusions of software experiments. We view the need for a variety of well-thought-out and well-designed software experiments as crucial if software metrics is to develop as a science.

There is a natural dichotomy in the interests of those who study software metrics. There are those whose interests lie in studies of the creation and management of programs -- in human performance. And there are those whose interests lie in studies of the objects produced -- in program performance. Although it is generally agreed that there ought to be a natural relationship between these two types of studies, we see no unifying theory developing in the near future.

Software metrics suffers at the moment in that virtually everything it studies is incommensurate with everything else. Consider how extraordinarily difficult it is to make a definitive statement comparing a parameter on software design, management, or construction if one is in APL versus if one is in COBOL. For the field to become more of a science there must be a dramatic increase in the number of precisely defined universally accepted software objects and parameters. But how do we arrive at such a desirable state? There is little doubt that the task would be significantly easier if there were one well-defined sufficiently powerful programming language used by all. But this has not come to pass. No one wishes to be constrained.

Since there is little likelihood that anyone will accept constraints on language, machine, and methodology merely in order to advance software metrics we must first study the relationships among the languages, machines, and methodologies we use and attempt to find the nature of the transformations between one and another. Otherwise the experiments, the data, the models, and the consequent theory of software metrics will have little value outside the realm of the laboratory.

Software seems almost infinitely malleable, and each individual change seems to require almost no effort. Yet to shape a piece of software to a precise end with no unforeseen consequences can paradoxically seem to require almost infinite effort. It is the task of software metrics to attach values and costs to every movement of software between these two extreme views.

The State of the Art and Research Recommendations

We now present an overview of the state of the art assessments and the near term research direction recommendations made in the papers which appear in this report. Three themes keep surfacing in the papers -- a need for more and better collected data, a need for a developing techniques for translating ideas and results from one domain to another, and a need for taxonomies of metric oriented terms and definitions.

In "Toward a Scientific Basis for Software Evaluation" Browne and Shaw explain how a great deal of energy is regularly invested in making measurements on software and its development process. The techniques of description, measurement, and evaluation are, however, mostly ad hoc. Most of the analysis and data collection techniques haven't been generalized beyond the local system for which they were developed. There is a general lack of a scientific foundation in this area on which to derive invariant principles from observations and experiments.

()

They recommend that future experimental work be based on the traditional principles of science where the first considerations are principles which are invariant across all software and hierarchies of abstract models. Although there now are several popular predictive models, they see more long term benefits coming from structural models of software. It is the development of this area for which they recommend support, with concentration first being on learning from structural models for specific systems.

In "Design of Software Experiments" Sayward summarized the principles used in conducting the many on going software experiments aimed at understanding and improving software development, testing, and maintenance. The designs of these experiments have nearly all followed the orthodox many subject random group design popularized by Sir Ronald Fisher. This area has led to the formulation of many interesting hypotheses. However, there are some basic problems with internal and external experiment validity which this area must overcome.

Sayward recommends continued support for the type of small scale many subject experiments which have been done since they will lead to interesting new hypotheses and they will produce a gradual refinement of design techniques for strengthening internal validity. He also suggests that a new approach, the single subject design, might be more natural for software experiments and recommends conducting intermediate scale single subject experiments as a potential way of strengthening external validity.

In "Data Collection, Analysis and Validation" Basili gives a variety of means for collecting data on software projects and he suggests approaches to dealing with the validity of the data. The means include the types of forms to have filled out and types of automated data collecting programs. Validity considerations include detecting incorrectly filled out forms and the detection of redundant data.

He recommends that more effort be put into establishing and refining a very large software lifecycle data base. It would be helpful to find ways of integrating the use of metrics into the data base categorization process. Establishing agreed on data collection terminology would also be helpful.

In "Experimental Evaluation of Software Characteristics and Programmer Performance" Curtis finds that most experimental studies on software metrics do not demonstrate cause-effects relationships between software characteristics and programmer performance. There are many uncontrolled factors which could have influenced the observed data. The biggest problem lies in replicating the environmental conditions under which real-world software is built.

Curtis recommends establishing long term multiple institute research programs which combine to replicate software experiments to see if the results are repeatable. He also recommends that experimental work be initiated on two forgotten areas -- programming language differences and the early lifecycle stages such as requirements and specification.

In "Software Forecasting" DeMillo and Lipton claim that the present searches for simple formulae for predicting the cost of large scale software efforts are very likely to fail. They explain how measurement theory rejects most of the formulae that have been suggested. On a positive side, they feel that the other popular method of predicting cost from historical data is more likely to produce success.

They recommend that predicting from historical data receive continued support and that more effort and thought go into establishing a better data base for these studies. They also recommend that an analogy from weather forecasting suggests a refinement which should be explored. That is, the development of micro theories of software costing (which don't necessarily scale up) and the development of large scale computational techniques (such as clustering) which integrate the micro theories to make a cost prediction for large systems.

In "Controlling Software Development Through The Life Cycle Model" Perlis states that almost no present systems are designed by taking a total view of the software lifecycle into account. Systems are designed to minimize design stage feedbacks, to localize the effects of maintenance, and to delay enhancements. No one ever thinks about making replacement easy. This view is what Perlis calls the "pre-structured approach." It assumes that there is but one pass through the lifecycle which, if all else is held constant, yields a perfectly designed and built system. Little attention is paid to building prototype systems -- they aren't necessary under this view.

Perlis recommends that a serious look be given to the sequence of executable prototype systems approach to software design as an alternative to the pre-structured approach. When coupled with the proper choice of language quite possibly this approach is more effective in dealing with the inevitable design changes, maintenance activities, and system enhancements.

In "Resource Models" Basili gives two approaches to predicting the resources (such as computer time, personnel, and dollars) needed for large scale software projects: formulating models and using historical data to estimate the constants in the resulting formulae; and, deriving equations from models of the problem solving process.

He recommends that experiments be conducted aimed at substantiating the formulae which have been derived. He also feels that research aimed at gaining insights on the software lifecycle from the derived formulae should be supported.

In "High Level Language Metrics" Sammet observes that high level programming language metrics are quite different and often confused with program complexity metrics. For comparing languages she categorizes two types of metrics: technical and non-technical. Among the former are feature counting and benchmarking. The latter include time to train personnel to be proficient in the language. The state of the art is that most of these techniques are more subjective than objective.

Sammet includes precise definitions of terms, quantitatively measuring the differences between languages, measuring the non-procedurality of languages, and measuring programmer productivity in a language among the most important research topics to support.

In "Performance Evaluation: A Software Metrics Success Story" Browne and Lynch find that performance evaluation of software systems is an area which has a set of generally accepted metrics for both external aspects (eg., response time) and internal aspects (eg., queue length). There is also a set of generally accepted abstract models which capture the salient concepts of performance and system tuning.

They recommend that support be given to a fundamental performance evaluation lack: software engineering procedures for developing software systems with desired values for given performance metrics.

In "Statistical Measures of Software Reliability" DeMillo and Sayward list two current scales of software reliability. First is the Boolean scale of formal verification, formal testing, and special programming disciplines. These don't really address a degree of reliability. The other is the continuous scale gotten by blindly applying hardware reliability theory to software. These attempts have led to unnatural and often contradictory assumptions. The basic problem is that software is not a fixed object and hence the distributional requirements of hardware reliability theory cannot be satisfied.

They recommend that, rather than striving for ways to assign a probability of correct operation to software, attempts to assign a probability to the processes used to validate software be supported. These validation processes are fixed over time and thus distributions for them can be studied empirically. Then a Bayesian based "level of confidence" in the correct operation of software validated by the process can be derived.

In "The Measurement of Software Quality and Complexity" Curtis explains that while there are many software metrics for measuring the product (programs) there are few metrics for measuring the process (programming). There have been several software experiments conducted on relating these two concepts in order that the product metrics can be used to predict the process time. Several interesting relationships have resulted, but it is far too early to accept them as laws.

Curtis recommends support for efforts to refine metrics, weeding out the redundant ones, and for efforts to validate metrics on larger data bases. Another important area of research is the development and validation of predictive equations.

In "Complexity of Large Systems" Belady finds that there are many speculative ideas around on the complexity of small programs but that few of them have been adequately tested. There is much duplicated uncoordinated effort. The two most important concepts for the complexity of large systems are evolutionary complexity and the time required to do a programming task. Little is known about evolutionary complexity.

Belady sees the need for a DOD coordinated effort to establish a large data base for validating a small standard set of complexity metrics. Also this data base could be studied in a search for patterns and trends in the evolution of large systems. This would lead to models of software evolution from which would emanate testable hypotheses. Another potentially fruitful study would be an exploration on the use of locality of information in large systems.

In "Software Maintenance Tools and Statistics in the Life Cycle of a Computing Application" Muller find that there is no real use of metrics in the maintenance lifecycle phase. Cost and profile data is usually gathered but there is no conceptual model of maintenance and hence no related set of software metrics has developed. The best things available are software tools such as the Programmer's Workbench which are aimed at easing bookkeeping tasks.

In order to initiate steps to improve this Muller list several research directions which include precise definitions of maintenance terms, a refinement of the maintenance data collection process, ways of detecting documentation deterioration, and ways of detecting errors introduced by maintenance activities.

In "A Scientific Approach to Statistical Software" Francis summarizes what has been done in evaluating statistical packages. The two questions of concern are software accuracy and cost of use. Quantitative measures have been defined for both questions and experiments have been conducted on the accuracy of these metrics. The experiments have relied on developing a standard set of problems.

Francis recommends an investigation into developing standard programs for the evaluation of metrics. He also recommends that a set of standard experiments be developed.

In "When is "Good" Enough? Evaluating and Selecting Software Metrics" Shaw examines the methods used by researchers to evaluate their proposed metrics and to compute the efficiency of metrics. Software metrics are either direct (eg., cost) or indirect (eg., time for cost). The majority of those found are indirect. Reasons for the proliferation of software metrics is that there is little attempt to find a basis set of direct metrics accompanied by models relating them to indirect metrics. Also metrics are applied without regard to precision or cost.

She recommends research into finding a small set of basis metrics which span most needs. This would avoid the syndrome of inventing a new metrics for each study and would entail finding models which relate direct and indirect metrics. Also recommended is a more critical use of classical statistics in evaluating metrics.

Toward a Scientific Basis for Software Evaluation

J. C. Browne

Department of Computer Science
University of Texas
Austin, Texas

Mary Shaw

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania

June, 1980

Abstract: An examination of the general practice of science, and in particular the interaction of experiment and analysis to generate structurally based system models, suggests a paradigm for the development of a science of software evaluation. We present a view of the development of structured models that is appropriate to software evaluation. We suggest research problems and research techniques which can lead to improvement in software measurement and evaluation methods.

1 The Problem of Software Evaluation

Most quantitative techniques for describing and evaluating software systems have been developed on a largely *ad hoc* basis. Large scale software development has been forced on us by the unprecedented speed with which computer systems have been integrated into the economic life of this country. The need for control over the software development process has created a "software engineering" discipline whose purpose is to establish operational procedures for the development of "quality" software. Unfortunately, techniques for predicting and evaluating software "quality" and performance have not yet emerged in practice. The purpose of this paper is to examine the intellectual tools and research attitudes that will be required before reliable metric techniques can be developed.

A great deal of energy is regularly invested in making measurements of software and of its development process. The techniques of description, measurement and evaluation are, however, largely *ad hoc*. Most of the analysis techniques and even the data collection techniques have not been generalized beyond the local system or application for which they were developed. As a result, large-scale comparisons of systems and evaluations at a level abstract from particular implementations are rare, and new measurement efforts can be expected to require the development of tools from scratch.

In other words, present metric techniques are not readily extended to new kinds of systems, new kinds of questions, or new development environments. As a result, there is no direct quantitative basis for comparing programs or software engineering methodologies. The economic ramifications

of this lack of extendibility are twofold. Each major software system requires a software engineering job starting almost from scratch. In many cases this engineering task is either slighted or totally ignored, leading to severe economic penalties in terms of effort, cost and delay. In addition, software engineering remains a high-technology discipline without adequate rules of standard practice; every system development requires expert skills and defies automation. This has serious economic significance, for the productivity of a group of technical personnel important to the economy is of paramount interest.

At present, software engineering is a technical activity for which we have developed a large set of ad hoc engineering techniques without a corresponding scientific foundation. We believe that this shortcoming is at the heart of many software engineering problems, including software metrics. There is substantial economic incentive for developing such a scientific basis for software evaluation. In Section 2 of this paper, we explore the proper relation between science and engineering and the role of modelling techniques in both. Section 3 discusses the role of scientific techniques and attitudes in software methodology. Section 4 examines current practice and points out discrepancies. Section 5 proposes steps to reduce the differences.

2 The Role of Models in Science and Engineering

We begin our description of the need for a scientific basis for software metrics by examining the related techniques of abstraction and modelling and by discussing the proper relation between science and engineering.

2.1. Complexity and Abstraction

Programs are not the only complex systems that humans must deal with. The national economy, for example, is far more complex; even the motion of the molecules in a simple physical object is many orders of magnitude more complex than the most complex program. However, no human *completely* understands the national economy or molecular structure in all their fine detail. In order to deal with complex situations we use a powerful technique -- abstraction. We deal with complex systems by ignoring their details; we develop *models* which reflect only certain important, macroscopic behavioral properties.

Newton's laws of motion, for example are a model of physical reality. This model abstracts from the enormous complexity of the motion of an object's component particles and describes gross properties of their aggregate behavior. For nearly all purposes the precise motions of the molecules in an object are irrelevant. The only relevant information is expressed by summarizing the individual motions in some way. For example, we speak of the velocity of the object (which is actually the average velocity of all its molecules), its temperature (a measure of the kinetic energy of the molecular

motion), and so on. When we summarize many details in a single property (velocity or temperature, for instance) we are *abstracting* from the details, or creating a *model* of the system.

For some purposes these gross abstractions may be inadequate. For example, the structure of a crystal depends on the properties of its constituent molecules. The abstraction used to describe crystalline properties must contain more detail than is used to describe the temperature of an object that happens to be a crystal. If we wish to understand chemical reactions we must consider an even more detailed model: the atomic structure of our materials. Notice something very important. In each of these cases we merely use another *model*, another *abstraction* of reality. Each model contains just enough detail to explicate the phenomena under study. To analyze the motion or temperature of an object, we can totally ignore its molecular structure. To analyze its simple chemical properties, we can ignore, for example, the wave-like behavior of subatomic particles. Only if we were to study nuclear reactions would we need a still more detailed model, namely quantum mechanics.

It is clear, then, that a deep philosophical assumption underlying modern science is that the complexity of reality can be understood by understanding a *hierarchy* of models -- some that describe macroscopic behavior by ignoring detail, others that successively explain increasingly microscopic behavior. Whether this simplifying assumption is entirely valid can be debated, but our limited intellectual capacity *forces* us to make it; without this assumption, we could not cope with the complexity surrounding us.

2.2. Science and Engineering

Engineering deals with the development and application of operational procedures for producing products or services. Science deals with models that explain relationships between the significant variables of systems. To do this, science must isolate and define the significant variables of the systems under study. These significant variables are usually identified by their appearance in statements of invariant relations or principles; system models established by science will incorporate these invariant relations. Engineering design of operational procedures is normally founded upon scientific models of systems.

Science usually starts with observations (measurements) and hypothesizes a set of principles or axioms to explain the observations. These axioms are used to derive or construct models of observable systems. The parameters or variables of these models may be derived from the axioms, or they may be estimated from observation. The model is then used to make new predictions about properties (metrics) of the observed system. The final step is to perform experiments (observations in controlled or understood environments) to determine the accuracy and robustness of models and of the statements of principles. The cycle of hypothesizing and validating models is then continued with the additional observations. Figure 1 illustrates this fundamental cycle of the scientific method; it is

important to note that the scientific analysis process of Figure 1 begins with measurements.

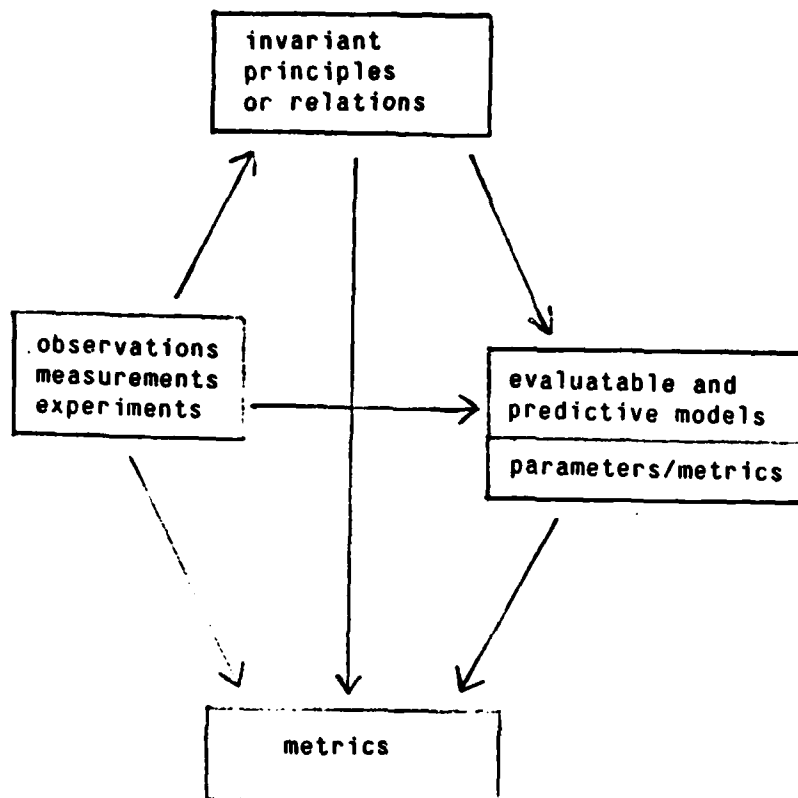


Figure 1. Model development cycle.

It is a crucial point that a single set of underlying principles generally serves as the basis for models of many systems in a broad variety of contexts. This modelling paradigm applies as well to the soft sciences as to the hard sciences. The difference is in essence that in the soft sciences such as psychology or social studies the problems are generally of a statistical or probabilistic nature.

Two kinds of models can be supported by measurements and statistics. The first, and the one on which we concentrate here, is the *analytic* or *structural* model, in which a system is described in terms of the way it is presumed to work and predictions about its behavior are derived from measurements or predictions about the behavior of its components. The second kind is the *descriptive* or *empirical* or *phenomenological* model, in which no attempt is made to model the underlying structure and predictions about system behavior are made by extrapolating from previous observations about the relation of system behavior to the values of various inputs. Structural models are most likely to be feasible for systems that are small enough for the investigator to make reasonable hypotheses about how they work. Descriptive models are more appropriate for large systems in which the underlying structure is not well understood, but for which it is important to be able to make some sort of

predictions. Although both kinds of models have legitimate roles in software engineering, good structural models are the mark of a mature engineering discipline. We therefore concentrate on the development process for structural models.

The determination of fundamental (metric) properties is an essential step in the formulation of a structured model. Fundamental properties are identified by their appearance in the invariant relations which unite observations across many environments; these invariants commonly express conservation relations. Identification of the fundamental properties is a major task of the model builder. However, instead of asking, "what are the fundamental properties of software systems?", we must seek to identify the invariant principles or relations which underlie experimental observations. The identification of fundamental properties or primitive metrics will emerge from the formulation of these invariants. We can then construct models for particular experiments, systems or methodologies with the guidance that the models must be consistent with the invariant principles, and they will depend on values of measurable inputs. These models will incorporate the fundamental properties as integral parts of their structure; the models may be constructive (software development methodologies) or analytic (mathematical or logical relations),

The invariant principles are guideposts and constraints in the development of model systems. In science, operationally useful models of physical systems are typically based on invariant principles. Newton's Laws are invariants for the mechanics of the material bodies of the normal environment; the fundamental laws of quantum mechanics -- the Heisenberg uncertainty principle and the DeBroglie relation -- are invariants for microscopic bodies. An active science has two threads of activity at each level of abstraction: one utilizes the invariant principles and their derived relations to study systems defined within the level of abstraction, and another attempts to resolve one level of abstraction with its upper and lower neighbors. Psychology is founded (loosely) on biology. Biology is based on chemistry. Chemistry is founded on physics, while physics seeks to resolve understanding to mathematical or symbolic relations.

The development of the gas laws in 19th century chemistry and physics illustrates both threads of development. In 19th century gas laws, Boyle's Law, the volume of a gas is inversely proportional to its pressure and Charles' Law, the volume is proportional to temperature, were entirely phenomenological. The development of Van der Waals' Equation which applies well to many real gases over a variety of conditions was based on kinetic-molecular theory. Quantum mechanics allows direct computation of Van der Waals' constants and indeed the precise virial equation of state. The conversion of mass to energy, for another example, occurs in many formats from solar fusion, to fusion reactors to internal combustion engines. The models which can be used to compute the energy output in each class of system are very different, but we can always be assured that the invariant principle of $E = mc^2$ will not be violated. It must also be recognized that the models of

science may vary enormously in resolution and detail. As an example, a model for the energy output of the sun may predict only total energy output or may be composed of flux into a wavelength spectrum and a kinetic energy spectrum for ejected particles on a local basis with respect to the sun's surface. These are further examples of the fundamental principle of hierarchical structuring. Similarly, models for software systems may legitimately be constructed to yield metrics at different levels of abstraction.

Both of the threads of understanding -- relationships within the field and the relation of the field to adjacent fields -- need to be pursued in the study of software: Software systems need to be understood in terms of appropriate metrics for the top level of concept abstraction, but it is also appropriate to try to map from programs to systems and from systems to programs.

3. The Practice of Science in the Context of Software Systems

We turn now to the application of this scientific paradigm to the analysis of software systems. This section discusses the lack of scientific foundations and sketches a suitable paradigm. The following section examines present practice in the field.

3.1. Lack of Scientific Basis

Software engineering has "leapfrogged" over the traditional engineering/science relationship. This "leapfrogging" has a number of consequences. The first and most obvious is that software engineering has been unable to produce broadly effective operational procedures for the development of large software systems with predictable characteristics. This assessment is not meant to decry or deride the enormous accomplishments of software engineering. On the contrary, there are few human endeavors since the pyramids where so much progress has been made in the face of so much difficulty and with so few tools in so short a time. It is simply not reasonable to expect that a given set of software engineering techniques can be extended beyond well-understood local environments without a basic scientific understanding of the structures and invariant principles underlying the systems involved. We do not yet have effective definitions of these underlying structures and principles for software systems, nor have we identified the significant variables by which software can be evaluated. Without a clear characterization of what is to be produced, we cannot hope to define an operational procedure for attaining the goal of producing software systems with desired characteristics. Another major impact of "leapfrogging" to engineering practice without a scientific basis is the confusion and intellectual disarray introduced into software engineering by the absence of basic scientific knowledge about the field. There has been a great deal of experimental and theoretical work on the creation and evaluation of methodologies for software development. This body of work, although carried out by careful and qualified practitioners, often contains

inconsistencies, ambiguities and contradictions. We attribute these problems to the lack of the unifying and integrating concept base which is developed through scientific principles of model building.

There is not now a well-established set of rules or concepts for analyzing or evaluating the properties of software systems. This is not to say that measurements of properties of programs and systems cannot be made; the problem lies in the fact that the variables which can be measured conveniently do not map readily upon properties or characteristics which are both quantifiable and comparable across a broad spectrum of contexts. Only properties or variables which satisfy both of the latter two requirements can be used reliably as metrics for evaluation of software systems. Since there is not a uniform basis for evaluating the software products we develop, it is difficult to make meaningful evaluations or comparisons of the methodologies used to produce these products. There is an excellent chance that the software measurements now being taken are not fundamental and that (even quantitative) descriptions of software do not refer to primitive and comparable concepts. There is an excellent chance that the relationships now observed and inferred in empirical models do not represent the actions of well-understood or fundamental mechanisms. In the absence of system-level metric properties or variables it is often the case that immediate metrics, which may be properties of languages or programming styles, are regarded as ends in themselves rather than pieces in the larger puzzle of a software system.

A given methodology may lead to software properties that are predictable by some metrics, but these metrics may not be relevant to products developed by other and competing methodologies. The next section discusses the traditional practice of the science in the context of software development and software evaluation problems. It is our contention that metrics for the evaluation of software can be developed only in the context of a scientific approach to the analysis of software.

3.2. Abstraction and Structure

Having argued that abstractions and models are essential, we now need to consider what constitutes a good abstraction or model. In the physical sciences the "Occam's Razor" principle has long been accepted as a criterion for choosing between competing models. In effect, Occam's Razor says "Pick the simplest model that adequately describes a phenomenon." The emphasis on simplicity is important, for the whole rationale for modeling is to accommodate our limitations in dealing with complexity. This is consistent with our use of simplicity as a criterion for good programming abstractions; in software design, too, we choose abstractions or models that are both simple (easy to understand) and yet adequate to describe the desired program behavior.

The classic Bohr atom is an excellent example of a simple model. There are only two kinds of parts: the nucleus and its orbiting electrons. Both parts can be characterized in terms of their weight

and charge, and the relation between them is characterized in terms of the simple laws governing orbital motion.

The notions of models and abstraction play a central role in software design, but the situation is somewhat different from the situation in the classical sciences. For the chemist and physicist, reality is fixed, given by the physical systems they study; they propose models to describe and predict reality. For them, the degree to which a model is successful can be tested by performing experiments and comparing the results to predictions from the model. The software engineer, on the other hand, is building an artificial reality. He can also use abstractions to control complexity, but he has the ability, in principle, to make these models exact by changing the reality they model. That is, he can (in principle) make the program perform at some level of abstraction exactly as predicted by the model under all circumstances. Thus, in a sense, the notions of modeling and abstraction are even more powerful in programming than in the physical sciences. Still, in practice the abstract descriptions used in program specification are almost always, to some extent, incomplete, just as they are in the physical sciences; for example, we seldom specify the exact running times of our programs.

Further, as the physical systems of chemistry and physics illustrate a hierarchy of models, each more detailed than the last, so is a similar hierarchical structure applicable to programming. An entire program is usually too complex to comprehend, so we describe it in terms of an abstract model. That model is defined in terms that, in turn, must be implemented with fairly complex programs. Generally we need to explain each of these subprograms in terms of still more detailed models. Thus, in a large program you should expect to find many levels of abstraction. Only in this way we can hope to avoid an explosion in complexity.

The illusion of malleability implied by this ability to, in principle, precisely control system behavior leads to problems in the development and maintenance of software. It also affects our present concern with measurement: The very malleability of the medium allows different models to be used for the development or analysis of very similar systems, and it allows the structure of programs to be changed by human design -- unlike physical systems, which are strongly constrained by natural laws. The implication of this lack of constraint on system organization is that any natural laws that operate must operate at a low level. Solid understanding of these primitive relations will be crucial to modelling large systems. In addition, the software metrician can re-emphasize the observation of many software builders that the illusion of malleability is misleading. Rigid rules governing system organization may well lead to more tractable systems.

3.3. Models for Software

A structural model explains a system in terms of component parts and presumed relations among those component parts. In the programming domain, the "parts" we deal with are data objects and

portions of programs. The relations between them are such things as how control and data values flow from one part of a program to another, what kinds of assumptions each portion of a program makes about its execution environment, and the ways the *humans working with the program* are organized. As scientists, engineers, and programmers, we strive to make both the individual parts and the relationships between them as simple -- as good -- as possible.

For example, if our overall objective is to measure "program quality", we might decide to characterize quality as a composite of a number of other, more specific, properties such as correctness, efficiency, maintainability, completeness, cost of development, quality of documentation, and so on. This reduces the problem of measuring quality to the subproblems of measuring the individual characteristics and of combining those results into an overall quality measure. We can address each of the subproblems in the same way -- we might, for example, work from a conceptual model that suggests maintainability depends on program structure, readability, size, number of authors, age, and so on. This generates yet another set of subproblems that could be addressed in the same way. Interestingly, some of these individual characteristics may reasonably be expected to contribute to the measurement of some of the properties of the original set: readability may also be a factor in cost of development and quality of documentation, program structure may also affect correctness and efficiency, size may also affect correctness and cost of development, etc. We speculate that a good structural model for program quality may in addition lead to better understanding of the factors that affect quality in programs. Severe problems are associated with making measurements such as these in such a way that they are valid across computers, programming languages, and even human organizations. The invariant relations that can help to normalize measurements across such differences have not yet been identified.

"Structure" is often cited as a "good" property or an indicator of the quality of a program. Several measures for program structure have been proposed. These include syntactic measures [6, 7, 12], counts of modules and modular connections in a program [1, 6], various properties of the graph of the program's control flow [10, 11], and the amount of information shared among modules [4]. Each proposal for a metric is supported by one or more tests or algorithms for collecting data to evaluate the metric. It would be remarkable if all these tests were measuring the same characteristic of a program; we have not yet seen adequate work on validating these proposed metrics, either individually or collectively. Although these studies do not view measures of structure as components of a more global metric methodology, we can imagine using them in that way. In order to accept "structure" as a measurable property, we must ask both about the pertinence of the proposed metrics to the property we call structure and about the accuracy with which the tests measure those metrics. We must also consider whether it is economically feasible to gather the data in practice.

Indeed, a given property may be measurable in more than one way. Certain aspects of operating

system performance are often measured by modelling the operating system as a simple single-server queueing system, and estimating the parameters of the model on the basis of empirical measurements [9]. To accept the results of such an analysis, we must consider both the degree to which the model represents the actual system and the accuracy with which the parameters are estimated. Operating system performance is also measured with benchmark tests [5]. A user of these tests must consider whether the jobs in the benchmark set correctly model the intended usage of the system and whether enough (and accurate enough) data is collected. Further, to the extent that queueing models and benchmarks are used to measure the same properties, we must be able to discuss how well they actually do so.

4. The Current Status of Experimental Science in Software Evaluation

Two aspects of the general methodology of science are missing from software metrics as currently practiced. These are the use of a hierarchy of models and the iterative cycle of hypothesis formation and validation.

The current cycle for experimental studies which should be establishing the foundation of a science of software evaluation is mostly conducted as shown in Figure 2.

Some quantifiable abstract property -- *metric* -- is selected for study (understandability, complexity, etc.). Some set of directly measurable quantities -- number of statements, reproducibility of code, etc. -- are selected. A model which relates the measurements to the metrics is assumed. The model is normally frankly phenomenological. The measurables are used in the model to generate values for the metric properties. These values are then commonly analyzed in an external context or correlated with other experiments. Sometimes the variables of the model are adjusted from analysis and correlation and new values of metric properties generated.

The well-known studies of program structure of McCabe [10] are an example. McCabe's work aims at establishing a property of programs which he labels complexity. Complexity is not given an abstract meaning or scale nor related to other properties at a similar level of abstraction. Rather, complexity is postulated to be a function of the program's control flow structure.

If this process is compared to the traditional processes of science it is quickly seen that an important step has been left out; the development of invariant principles from observations and experiments. Programs unfortunately lack characteristics that can readily be quantified or expressed on a linear numerical scale. For example, control structures are not entirely comparable across programming languages even when extracted to graph structures. There is no intrinsic quantification of program complexity; it is regarded as a *derived* quantity.

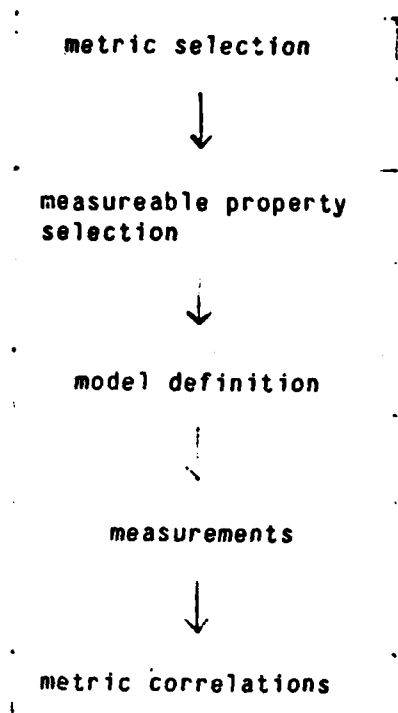


Figure 2. Current Practice for Experimental Software Science.

A second shortcoming of current software practice is the lack of attention to hierarchical structure in metrics. The paradigm of science suggests a search for invariant relations involving complexity and attempts to *quantify* complexity in absolute terms or to relate it to other properties at similar levels of abstraction. Such relations could then be used to calibrate models with intermediate, perhaps not comparable, variables. The absence of a foundation of first principles renders the model entirely empirical. Such a model may have predictive value in a given situation, but it is not likely to contain enough structural information to be extendable to other situations. It will be very surprising for coherence and consistency to develop in experimental studies of software development until the field develops some basis of invariant principles. It is not necessary that the invariant principles be rigid mathematical structures; they may be expected to be statistically probabilistic for those aspects of software system evaluation which involve human judgment or human usage.

Models for productivity have often been developed purely empirically. For example, Walston and Felix [13] develop equations for several aspects of programmer productivity based on the assumption that equations of the form

$$\text{EFFORT} = a \cdot \text{SIZE}^b$$

adequately model the situation. Based on empirical data, they obtain equations for a number of characteristics of productivity, including

$$E = 5.2 L^{0.91}$$

$$D = 49 L^{1.01}$$

where E = effort in man-months

L = thousands of lines of delivered code

D = pages of documentation

In discussing the first equation, they remark that it is nearly linear, but they do not hypothesize a structural explanation and they do not attempt to revise the model to express effort as a linear function of code size with second-order effects. In another paper in this report, Basili describes an attempt to apply the Walston-Felix model in a different environment. He finds that historical data for the second environment yields equations including

$$E = 1.41 L^{0.94}$$

$$D = 29.5 L^{0.92}$$

These are again close to linear, but again the linearity is noted but not incorporated in the model. Basili also notes that Lawrence and Jeffery [8] have successfully used a linear model. These models for productivity serve an immediate role in providing predictive information for cost planning of current development projects. However they do not serve to illuminate the basic causal relations that underly the model.

The work of Walters and McCall [14] on the development of metrics for software reliability and maintainability is an example of careful and thorough analysis which proceeds by the entirely phenomenological model represented in Figure 2. They, in fact, go beyond typical current practice by including a hierarchically structured model. They postulate that reliability and maintainability are functions of other more resolved concepts of programs, modules and systems. Reliability is postulated as deriving from error tolerance, consistency, accuracy and simplicity. Maintainability is said to derive from consistency, simplicity, conciseness, modularity and self descriptiveness. These assumed metric properties of modules and systems are in most cases evaluated for each module and for the entire system. Values for a top level metric such as maintainability are derived from independent measurements. Linear regression fits between the metrics at the top level and the module system metrics are assumed and coefficients determined by regression analysis. This paper found significant correlation between values of top level metrics and the values of module/system metrics such as effectiveness of comments, complexity, etc. This work is thorough and professional. It is, however, deficient with respect to the traditional practice of science in two regards. The model structure is assumed. There is no effort to determine other than linear relationships between metrics or model parameters. There is no attempt to look beyond correlations for invariant principles or to determine further model structures. It is these latter two aspects which we feel must be added to

observation/correlation in order to lay a scientific basis for software evaluation metrics.

The absence of a basis of invariance principles not only hinders the formation of coherent models or relations between the variables of a system, it also hinders the specification of measurement. The invariant principles are relations between fundamental quantities. It is often difficult to discern what properties are fundamental in the absence of the guidelines given by invariant principles. It is thus difficult for current experimental work to pose questions (hypotheses) in such a way that after the experiment it can be told whether or not the questions have been answered (tested). The situation is that persons doing experiments in software evaluation are faced with the circumstances of not having the basis for understanding what it is that they are supposed to be studying while trying to design and execute the experiments giving values for these quantities. They face the ultimate problems of experimentation -- to demonstrate cause-and-effect relationships without having firmly based definitions of what is to be a cause and what effect is to be measured.

There are currently two main threads of experimental work. One is the direct analysis of software systems' measureable properties and properties presumed to be fundamental. Another is concerned with the evaluation of methodologies for software development in terms of the application of the methodology producing predictable values for assumed metric properties. These two streams of work now tend to use different metrics and to do generally incompatible experiments. A strong effort needs to be made to merge these two paths if true experimentation which implies operational control, the ability to generate cause and effect relationships in experiments, is to be developed. The development methodologies need sound definition of metrics and structural models for a sound base while experiments in software science must of necessity be constructive in nature.

The validation of models in this situation poses substantial difficulties. In the current structure, validation consists of showing that the given structure in a given set of variable values allows the model to map input to output in at least one case. This is a powerful validation procedure only if the model can also be validated to conform to invariant principles which are known to be valid for a wide spectrum of system descriptions and if the model can be shown to generally conform to invariant structures.

The concept of successive refinement is not often invoked in current experiments in software evaluation. Rather, the intermediate metrics are often regarded as an end in themselves.

Response time analysis of computer systems provides an example of a software evaluation experiment which conforms to a traditional paradigm for scientific analysis. Response can be measured as functional of transaction resource usage characteristics. A queueing network model representation of an interactive computer system might be as in Figure 3, taken from [2]. Response time is modeled on the time elapsed between a job leaving the TERM queue and rejoining the TERM

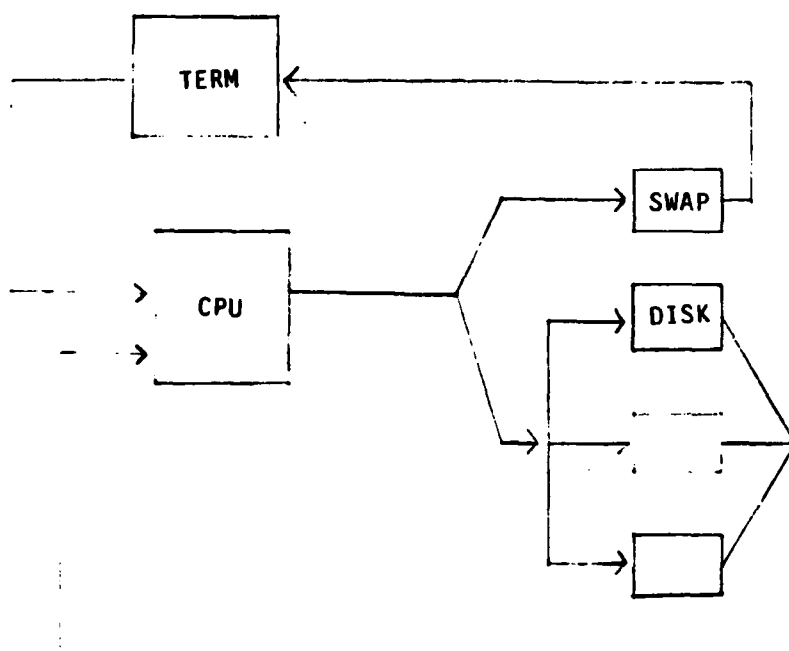


Figure 2. Queueing Network Model.

queue. The disk system is actually a two phase process, positioning and transfer. Additionally, each disk sub-system probably consists of several drives on a single controller. Therefore each DISK queue is really a composite structure such as Figure 4. The service time for POSITION is normally a function of workload level. Measurements are made on each phase of the disk service and the model is parameterized as a function of workload. The Chandy-Herzog-Woo (CHW) theorem [3] is then used to determine an equivalent single queue to represent DISK in Figure 3. Figure 3 is then collapsed to Figure 5 by the CHW theorem and response time becomes the service time of the queue labelled SYSTEM. The model relationships are determined by the structure of the several queueing network models and the parameters are taken directly from observations and invariant principles.

5. Steps Toward a Scientific Basis for Software Evaluation

The comparison of the successful development pattern for the sciences with the current ad hoc practice in software evaluation suggests directions for both long term and short term research and development. The current empirical and experimental work is vital. It should be enhanced and extended by applying the general principles of traditional science.

1. *Invariance Principles* - The absence of invariance principles is the most serious lack in software evaluation. Invariance principles are seldom created. Rather they are synthesized from observation and from transfer of knowledge coupled with insight and a knowledge of the systems being studied. Thus analysis of observed properties of

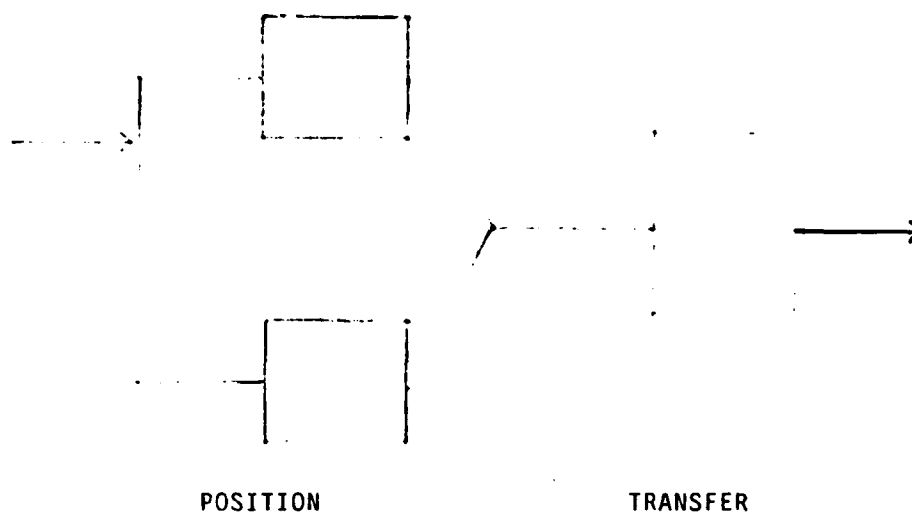


Figure 4. Disk System Model.

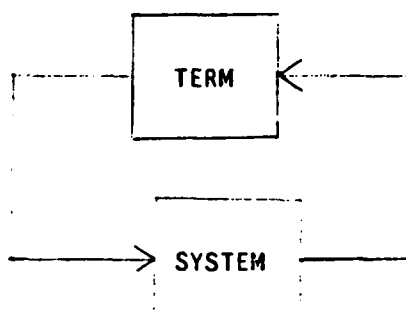


Figure 3. Collapsed System.

software is not wasted effort. It is almost surely a prerequisite for eventual generation of invariance principles. Much more emphasis needs to be placed on attempts to identify abstract invariance principles from observations rather than the current attempts to create entirely empirical context driven models.

2. *Successive Refinements* - The concept of hierarchical structuring of metrics and models needs to become a part of the experimental practice of software evaluation. The attempts to leap the vast conceptual distance from concepts such as maintainability directly to properties of programs written in the specific languages is unlikely to produce success. Experiments and models need to be constructed which attempt to define and evaluate such programs or system concepts such as maintainability, etc.
3. *Model Validation* - Model validation must be recognized as a principal problem. Efforts need to be made to test models against even assumed invariance principles. The definition of the context of a validation and the precision of definition of the metrics and the variables in the model need to be an integral part of every experimental study. Phenomenological and empirical models need to be analyzed for possible content of and effect of invariant principles. The principle of successive refinement is important with

respect to the validation problem. The decomposition of the concept goals into intermediate metrics leads to the possible establishment of intermediate levels of validation which may be soundly based and perhaps well understood.

4. *Introspection* - We must iterate over our models to make them better descriptions of the systems they model. This is different from creating a hierarchy of successively more precise models - it is rather a requirement for better accuracy at each level of precision.

There is a strong need for experimental work based upon these traditional principles of science. A few such experiments, if successful, could lead to fundamental new directions for software system evaluation and software engineering. Although purely predictive models will be useful, and they may yield higher cost returns in the short run, structural models provide a means for understanding the fundamental mechanisms at work in a system and for basing new models on existing sub-models. As a result, structural models offer substantial economic benefits in the long term. We recommend that investigations of techniques for developing, evaluating, and using structural models be supported as basic research. For the next few years, this must include support for the development of models for specific kinds of systems to serve as test beds and concrete examples of good practice.

6. References

- [1] L. A. Belady and C. J. Evangelisti.
System Partitioning and its Measure.
Technical Report RC 7560 (# 32643), IBM T. J. Watson Research Center, March, 1979.
- [2] J.C. Browne, K.M. Chandy, R.M. Brown, J.W. Keller, D.F. Towsley, and C.W. Dissly.
Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems.
Proceedings of the IEEE 63:966-977, 1975.
- [3] K.M. Chandy, U. Herzog, and L. Woo.
Parametric Analysis of Queuing Networks.
IBM Journal of Research and Development 19:36-42, 1975.
- [4] Robert N. Chanon.
On a Measure of Program Structure.
Technical Report, Carnegie-Mellon University, 1973.
- [5] Dennis M. Conti.
Findings of the Standard Benchmark Library Study Group.
Technical Report 500-38, National Bureau of Standards Special Publication, January, 1979.
- [6] Tom Gilb.
Software Metrics.
Winthrop, 1977.
- [7] Maurice H. Halstead.
Operating and Programming Systems: Elements of Software Science.
Elsevier Computer Science Library, 1977.

- [8] D. R. Jeffery and M. J. Lawrence.
An Inter-Organizational Comparison of Programming Productivity.
Technical Report, University of New South Wales, Department of Information Systems, ??.
- [9] L. Kleinrock.
Queuing Systems, Volume II: Applications.
John Wiley & Sons, 1975.
- [10] Thomas J. McCabe.
A Complexity Measure.
IEEE Transactions on Software Engineering SE-2(4), December, 1976.
- [11] Barbara G. Ryder.
Constructing a Call Graph of a Program.
IEEE Transactions on Software Engineering SE-5(3):216-226, May, 1979.
- [12] T. A. Thayer, et al.
Software Reliability Study.
Technical Report RADC-TR-76-238, Rome Air Development Center, August, 1976.
- [13] C. E. Walston and C. P. Felix.
A Method of Programming Measurement and Estimation.
IBM Systems Journal 16(1), 1977.
- [14] Gene F. Walters and James A. McCall.
The Development of Metrics for Software Reliability and Maintainability.
In Proceedings of the 1978 Reliability and Maintainability Symposium. 1978.

Design of Software Experiments

Frederick G. Sayward

Department of Computer Science
Yale University
New Haven, Connecticut

June 1980

I. Introduction

In this paper the current practices used in conducting experiments aimed at understanding and improving software development, testing, and maintenance will be summarized and critiqued. To date the designs used in software experiments have followed the traditional approach in which subjects are randomly drawn from a population and then randomly assigned to two or more groups. The effect of the hypothesized improvement factor is then evaluated by comparing the mean intergroup change observed on some measured factor. In addition to the problems of internally validating current software experiments, external validation problems have led to little wide spread acceptance of their conclusions. The chief research directing recommendation given is that a relatively new and still controversial experimental design, called single subject research [16], should be explored as a possible cost-effective approach to the external validation problems of current software experiments.

In the next section the principles and terminology of experiments will be reviewed. A summary and analysis of the methods and conclusions of software experiments found in the literature is given in section 3. In section 4 the principles and problems of single subject experiments are given along with an indication of how this design type might be

applied in software experiments. Suggested research directions are presented in the final section.

II. Experimental Principles and Terminology

Experiments of the type conducted for understanding and improving software usually start with an identification of three types of variables: independent, dependent, and uncontrolled. The independent variable is the hypothesised improvement factor (treatment). An example of an independent variable is using or not using structured programming techniques during program development. A dependent variable is some measurable factor (response) which is a function of the independent variable. For example, the cost of program development. Uncontrolled variables are any other factors (sources of variance) which influence the dependent variable. For example, years of programming experience, programming language, and program size.

Hypotheses are usually stated in terms of these variables. For example, "structured programming reduces the cost of program development" and "the cost of developing 500 line FORTRAN programs is reduced by a factor of 3 when structured programming is used." Clearly, the more general the hypothesis the more effort required to design a valid experiment.

The next aspect is choosing a design type. The most popular choice in software experiments is the many subject random group design which has evolved over the past hundred years from work initially done in agricultural experiments. After random selection, the subjects are randomly split into groups. Both uses of randomization are critical in

accounting for the dependent variable measurement variance introduced by some of the uncontrolled variables. Also, groups are assigned instances of the independent variable for the duration of the experiment.

After subject selection and assignment, the material of the experiment must be determined. For software, this is usually the programming tasks to be done. Depending on the hypothesis, randomization may again be used to dampen variance.

The next decision is in the measurement technique used to monitor the dependent variable. For example, it may be too difficult under experimental conditions to measure cost in dollars. But if one makes the inference that time is money, then one is justified in measuring time. In the many subject random group experiment, measurements on the dependent variable are (effectively) taken only once, that being at the conclusion of the experiment. A single measurement is also in the agricultural tradition where crop yield is the critical data. This point should not be taken lightly since a large body of statistical inference tests have evolved for analyzing this single (in time) data point. These tests form the analytic tools which have been used to analyze the data collected in software experiments in determining whether the hypothesis should be accepted or rejected.

The final, and probably most important aspect, concerns experiment validity. There are two forms of validity; internal and external. Internal validity is the degree of certainty that changes in the independent variable account for observed changes in the dependent variable. That is, all uncontrolled variables have been ruled out as contributors to the measurements taken on the dependent variable.

External validity is the extent to which the results of the experiment can be generalized to different subjects, materials, statistical tests, and experiments. In general, do the experimental results scale to the entire population? Internal validity is a minimum prerequisite for conducting an experiment since without it the results of an experiment are uninterpretable. For many subject random group experiments, there are several replication methods which have been used in the design of software experiments, such as factorization and Latin squares, aimed at ensuring internal validity.

III. State of the Art in Software Experiments

As stated above, the design of software experiments has followed the traditional many subject group orientation. The independent variables under investigation have fallen mainly into the following five categories: programming language constructs, programming methodologies, program complexity measures, types of debugging tasks, and programmer experience. The dependent variables have fallen mainly into the following four categories: time and accuracy of program development, understanding of existing programs, number of detected errors, and time and accuracy of modifications to existing programs. Depending on the study, some of the above factors are taken to be uncontrolled variables. It is noted that there is a surprising lack of using commonly used programming languages as an independent variable. Programming language usually is a fixed parameter in software experiments.

The number of papers on software experiments has grown rapidly in the past few years indicating interest both in the computer science research and in the practitioner camps. What follows is a brief summary of the software experiments literature examined by the author, presented more or less in chronological order. Unless otherwise stated, each experiment used some form of many subject random group design.

Sackman, Erikson, and Grant [20] found that on-line debugging took less time than off-line debugging with no significant increase in CPU time.

Sime, Green, and Guest [28] found that the IF THEN ELSE type of conditional statement is superior to the IF THEN GOTO in terms of programming time and number of errors made.

Gould and Drongowski [14] found up to a factor of five difference in the time required to detect different types of bugs, and found up to a factor of four difference in the number of bugs detected by experienced programmers having equivalent backgrounds.

Weinberg and Schulman [30] found that programmers construct quite different software to satisfy the same objectives of fast programming, efficient programs, minimal program size, and program readability.

Youngs [32] found that experienced programmers use different debugging strategies than those used by beginning programmers.

Weissman [31] found that commenting and paragraphing made programs easier to understand but more difficult to modify while good variable naming helped both program understanding and modification.

()

Gould [13] gave examples of debugging tasks and program modifications which require little to no understanding of the programs under consideration.

Shneiderman [21] found that using FORTRAN logical IFs rather than arithmetic IFs made small programs easier to understand for novice programmers.

Gannon [10] found that strong data typing in programming languages has positive effect over no data typing respect to making fewer errors and taking fewer runs in arriving at the final program.

Green [15] found that professional programmers were better able to answer difficult questions about programs written in a language which used an IF THEN ELSE having redundant predicate information rather than the standard IF THEN ELSE.

Shneiderman [22] found that even a few months difference in experience for intermediate level programmers can have a significant effect on performance and he also advocated the use of memorization and recall as a basis of measuring program understanding.

Shneiderman, Mayer, McKay, and Heller [24] found that flowcharts are redundant and have a potential negative affect on coding, comprehension, debugging, and modification.

Sime, Arblaster, and Guest [26] found that tools for helping format conditional statements reduced the initial program error content but didn't help in locating the error initially present.

Sime, Green, and Guest [29] found that the IF THEN ELSE type of conditional statement is superior to the IF THEN GOTO in terms of programming time and number of errors made.

Brooks [3] found that dictionaries of program variables are superior to macro flowcharts as an aid to understand program control and data structures.

Chrysler [5], using no controlled groups, found that for COBOL programs the occurrence counts of output fields, input files, control breaks and totals, input edits, output records, input fields, and input records were the most significant predictors of development time and that programmer age and years of formal education were the most significant predictors of development time.

Myers [18], in comparing different debugging methods, found that surprisingly few errors (34%) were caught, there were significant differences in the error types caught by group, and that code walkthrough was more costly than other methods but didn't give better performance.

Basili and Reiter [1] found that three man teams, in building a simple compiler, which were required to use modern programming methods required less effort than uncontrolled teams or individuals.

Curtis, Sheppard, Milliman, Borst, and Love [7], in comparing Halstead's effort metric, McCabe's cyclomatic number metric, and the number of executable statements to programmer performance on two maintenance tasks, found there was little to choose in the three metrics with respect to time and accuracy of maintenance.

Curtis, Sheppard, and Milliman [6] in a replication of [7] using larger multimodular programs and a wider variety of subjects found that Halstead's effort metric was better than lines of code by a factor of two for predicting debugging effort.

Dunsmore and Gannon [8] found that average variables referenced per statement and average live variables per statement had a simplifying effect on program development and program maintenance.

Shneiderman and Mayer [23] found planned modularization rather than no or random modularization aided in the comprehension of programs.

Schneidewind and Hoffmann [25] found that the rates for making programming errors, detecting programming errors, and correcting programming errors are all dependent on structural program complexity.

Sheppard, Curtis, Milliman, and Love [26] found that one could use the number of languages known and the familiarity with FORTRAN concepts to predict comprehension, modification, and debugging performances for programmers having three or less years of FORTRAN experience, but not for programmers having more FORTRAN experience.

Dunsmore and Gannon [9], in investigating the effect program nesting, percentage of global variables, ratio of parameters to global variables, average variables referenced, and average live variables have on the effort required for program construction, comprehension, and modification, found that effort related to all five parameters, and that modification is easier when the ratio of formal parameters to global variable is high and when the average live variables per statement is low.

()

Rarely in the above software experiments have hypotheses been formally stated at the outset. Rather, the paradigm is an informal introduction to the factors under consideration, the experiment design, the data collected, the application of statistical inference tools, and conclusions. The reported internal validity considerations are rarely satisfactory. Each paper usually ends with a note that the findings suggest that further investigation should be done, indicating that even the authors have little faith, as yet, in the external validity of their software experiments.

In [4], Brooks gives a detailed account of some internal validity flaws found in current software experiments. He cites several examples of subject and material selection, summarized below, which make many experiments suspect. With respect to measurement selection, he states that new measures of the effort required for program construction and program understanding based on cognitive models of program-programmer interaction are needed.

Brooks' criticism of subject selection is based on the methods used to circumvent the problems of cost, non-availability and wide ability differences of using experienced professional programmers. Most software experiments have used beginning or intermediate level student programmers as subjects. There is little proof of the necessary internal validation issue that experienced programmers use at a faster rate the same problem solving procedures as do beginners. Also, guaranteeing that groups of beginning programmers have equal ability is not trivial.

The problem with material selection is not so much internal as external. For internal validity, the programming tasks selected must be comparable across the uncontrolled variables. Program complexity measures have been somewhat helpful in this respect. Factorization designs can also be used.

For external validity, Brooks states that the programs used in software experiments are not representative of the programs being developed in the real world because of their small size. Since it is universally accepted that developing a large system is not just a matter of scaling up from developing a small system, small scale material will be a potential source of external invalidity until an accepted model of the effects of program size is developed.

Brooks states that all of the experiments done to date need to be replicated on larger programs before any generality of their results can be accepted. Although not stated, presumably experienced professional programmers would also have to be used in attempting to generalize the current software experiments by replicating them on large scale software.

It is the opinion of this author that Brooks' suggestion is not only economically infeasible but also premature. In the next section a more economical but equally risky alternative suggestion will be made which is based on the following question:

Given that the software life cycle is so dynamic over time, should we be content in our software experiments to draw conclusions based on a single ex post facto measurement?

IV. Single Subject Research

The major difference between the orthodox many subject experiment and the single subject experiment is that in the latter approach the dependent variable is measured throughout the experiment to yield a time-series of observations. In their design, time is broken into phases during which different instances of the independent variable are applied (called interventions). Conclusions are then drawn from observed interphase changes in the dependent variable.

There are four basic types of single subject experiments: one individual, two or more groups of one individual, one (small) group of subjects, two or more (small) groups of subjects. Hence, single subject is really a misnomer. Within each type of experiment there is further diversity in the number and order of interventions. However, in all designs the emphasis is on a time-series of measurements.

As usual, the design problems are in internal and external validity. It is more difficult to resolve the internal issue for single subject than for many subject experiments. Several recommendations on validity are given in [16] and some will be summarized below.

For now, some of the validity problems will be illustrated by considering a hypothetical single subject experiment on the hypothesis "structured programming reduces the cost of program development." The single subject design will be of the operant or ABAB type which is used for one individual or one small group. For illustration purposes, let us assume we have a perfect definition of structured programming and a perfect measure of program development cost which can be applied in an unbiased fashion at any point in time.

In the ABAB design the time axis is broken into four phases whose sequence and terminology are illustrated in figure one.

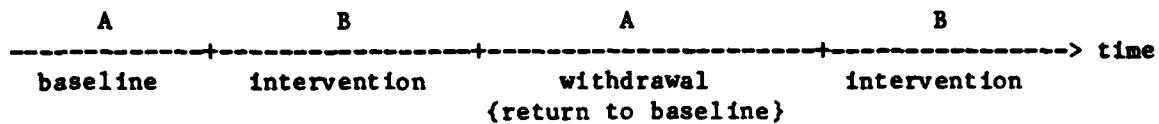


Figure One: The ABAB single subject design.

During baseline and withdrawal the single subject does not use structured programming techniques, while in the other two phases structured programming is employed. Suppose that the subject is writing a large system in some language and periodically the cost measure is applied to arrive at the raw data which is graphically presented in figure two.

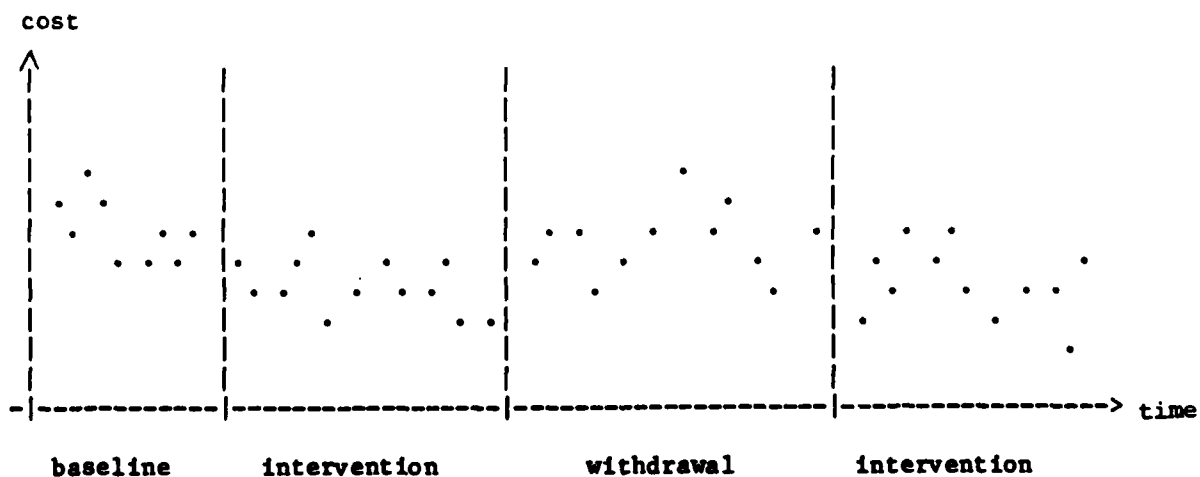


Figure Two: Hypothetical raw data.

As will become evident, graphical presentation plays a central role in interpreting the data of single subject experiments. In [19] several methods of constructing line graphs, bar graphs, and cumulative graphs from the raw time-series data of a single subject experiment are given.

Also, useful graph transformations are given. It is explained how the various graphical forms can be used as an aid in analyzing several experiment validity questions.

Two important graphs are illustrated in figures three and four: a bar graph of the phase means and a within phases least-squares fitted trend line.

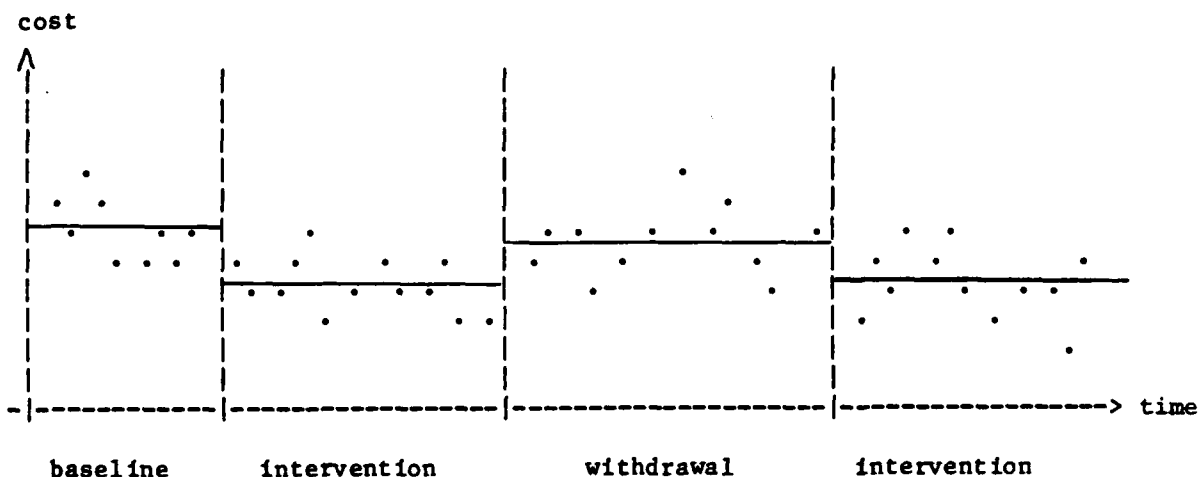


Figure Three: Bar graph of phase means.

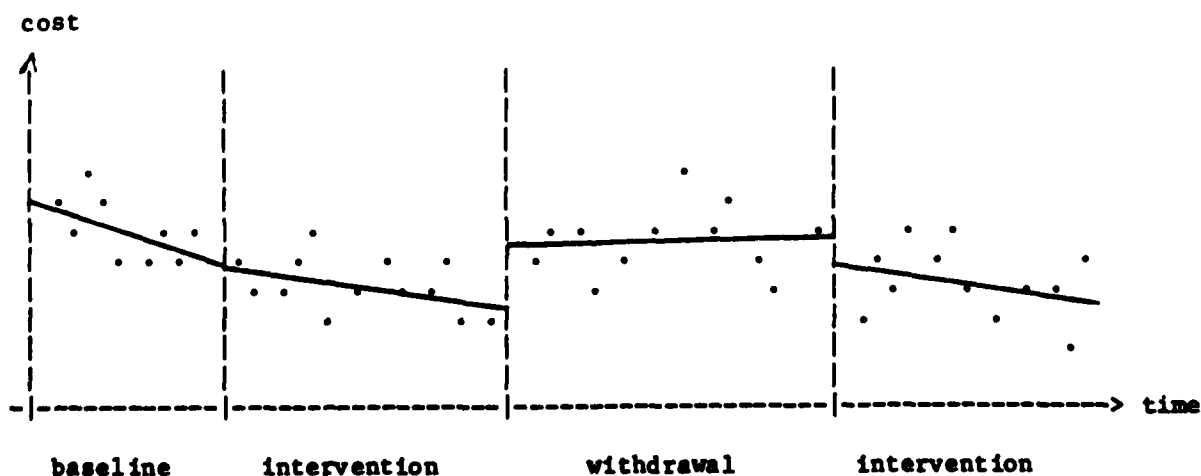


Figure Four: Least-squares fitted trend lines.

Figure three indicates a necessary for internal validity equal levels of better performance by the subject when structured programming is used. However, since the subject performed better under withdrawal than at baseline, there is a threat to internal validity due to learning. This is compounded in view of the decreasing baseline trend line illustrated in figure four. Thus, there is a major threat to validity due to an unstable baseline. A prerequisite in single subject experiments is the establishment of stability in the baseline. Although not illustrated, variance in the phase data must also be considered.

In [17] Kratochwill gives a summary of several validity issues which must be dealt with in single subject research. The threats to internal and external validity for single subject experiments are more or less the same threats present in many subject experiments. But the problems in dealing with the threats are more acute due to two facts: single subject experiments usually take more time to conduct and thus there is greater chance that a threat causing event will occur; and since there are fewer subjects, subject programs such as learning or drop out have greater negative impact on validity. In general, more attention must be paid to validity in single subject experiments since the design is less adequate for dealing with the influences of the uncontrolled variables.

On the positive side, the problem of representative material is somewhat easier. Also there are two emerging formal methods are given for data evaluation: graphical techniques [19] and statistical inference techniques [11,12]. In the latter area it has been shown how time-series analysis and Markov chains can be used as significance tests for baseline stationarity, interphase changes in level, and autocorrelation in the collected data.

V. Recommended Research Directions

The chief research directing recommendation of this paper is that computer scientists engaged in software experiments should seriously consider using single subject designs in their subsequent experiments. Unlike the agricultural tradition of many subject designs, single subject designs have evolved from an educational psychology and therapeutic tradition. In these fields the difficulties of replication and the ethics of non-treatment led to a search for alternative designs. While the latter point doesn't appear to be a problem in computer science, the former certainly is.

A major potential advantage of single subject research to software experiments lies in retesting hypotheses which have been positively affirmed in past experiments. As stated above, these affirmations are often suspect due to the small scale material used and the selection of inexperienced subjects. Added strength can be gained through single subject experiments using larger scale material and more experienced subjects. Although these same goals could be accomplished by scaled-up replication of past experiments, as suggested by Brooks [4], it is not felt that there is adequate justification for the staggering costs this would entail at the present time. Also, some of the threats to internal validity would still be present in the replications since there is, as yet, many deficiencies in our knowledge of programmer-programming task interactions. In summary, single subject design offers a cost-effective opportunity do to larger scale software experiments now.

From a philosophical view, the single subject approach is more appealing to this author because of its data collection over time mandate. Given that the software life cycle is very dynamic, it is difficult to accept hypotheses on the basis of data collected at a single point in time. Even if the validity issues of single subject software experiments prove too difficult to overcome, there is a good chance that conducting such experiments will lead to new insights on how to integrate the time variable into traditional software experiments.

Lastly, in addition to supporting the new single subject approach, it is recommended that small scale many subject software experiments receive continued support. These experiments will continue to lead to interesting new hypotheses, a sharpening of subject, material, and measurement selection techniques, and a quick inexpensive rejection of false hypotheses. Integrating the recommendations, one can foresee the future possibility of hypotheses passing through a hierarchy of well planned software experiments: small scale single subject, small scale many subject, intermediate scale single subject, intermediate scale many subject, ...

Acknowledgment

I would like to thank Richard Lipton for making me aware of single subject research and for introducing me to an invaluable reference [17].

References

- [1] V.Basili and R.Reiter, "An Investigation of Human Factors in Software Development", Computer 12,12, Dec. 1979, pp. 21-38.
- [2] S.Boies and J.Could, "Syntactic Errors in Computer Programming", Human Factors 16, 1974, pp. 253-257.

- [3] R.Brooks, "Using a Behavioral Theory of Program Comprehension in Software Engineering", Proc. 3rd Internat. Conf. Software Eng., IEEE, New York, 1978, pp. 196-201.
- [4] R.Brooks, "Studying Programmer Behavior Experimentally: The Problems of Proper Methodology", Comm.of the ACM 23,4 (Apr. 1980), pp. 207-213.
- [5] E.Chrysler, "Some Basis Determinants of Computer Programming Productivity", Comm.of the ACM 21,6 (June 1978), pp. 472-483.
- [6] B.Curtis, S.Sheppard, and P.Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", Proc. 4th Internat. Conf. Software Eng., IEEE, New York, 1979, pp. 356-360.
- [7] B.Curtis, S.Sheppard, P.Milliman, M.Borst, and T.Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", IEEE Transactions on Software Engineering SE-5, March 1979, pp. 96-104.
- [8] H.Dunsmore and J.Gannon, "Data Referencing: An Empirical Investigation", Computer 12,12, Dec. 1979, pp. 50-59.
- [9] H.Dunsmore and J.Gannon, "Analysis of the Effects of Programming Factors on Programming Effort", The Journal of Systems and Software 1, 141-153 (1980).
- [10] J.Gannon. "An Experimental Evaluation of Data Type Conventions", Comm.of the ACM 20,8 (Aug.1977), pp. 584-595.
- [11] J.Gottman and G.Glass, "Analysis of Interrupted Time-Series Experiments", in T.Kratochwill (Editor), Single Subject Research: Strategies for Evaluating Change, Academic Press, New York, 1978, pp. 197-235.
- [12] J.Gottman and C.Notarius, "Sequential Analysis of Observational Data Using Markov Chains", in T.Kratochwill (Editor), Single Subject Research: Strategies for Evaluating Change, Academic Press, New York, 1978, pp. 237-285.
- [13] J.Gould, "Some Psychological Evidence on how People Debug Computer Programs", International Journal of Man-Machine Studies 7, 1975, pp. 151-182.
- [14] J.Gould and P.Driongowski, "An Exploratory Study of Computer Program Debugging", Human Factors 16,3, 1974, pp. 258-277.
- [15] T.Green, "Conditional Program Statements and Their Comprehensibility to Professional Programmers", Journal of Occupational Psychology 50, 1977, pp. 93-109.

- [16] T.Kratochwill (Editor), Single Subject Research: Strategies for Evaluating Change, Academic Press, New York, 1978.
- [17] T.Kratochwill, "Foundations of Time-Series Research", in T.Kratochwill (Editor), Single Subject Research: Strategies for Evaluating Change, Academic Press, New York, 1978, pp. 1-100.
- [18] G.Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/ Inspection", Comm.of the ACM 21,9 (Sept.1978), pp. 760-768.
- [19] B.Parsonson and D.Baer, "The Analysis and Presentation of Graphic Data", in T.Kratochwill (Editor), Single Subject Research: Strategies for Evaluating Change, Academic Press, New York, 1978, pp. 101-165.
- [20] H.Sackman, W.Erikson, and E.Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", Comm.of the ACM 11,1 (Jan. 1968), pp. 3-11.
- [21] B.Shneiderman, "Exploratory Experiments in Programmer Behavior", Int'l J. Computer and Information Sci., 5,2 (June 1976), pp. 123-143.
- [22] B.Shneiderman, "Measuring Computer Program Quality and Comprehension", International Journal of Man-Machine Studies 9, 1977, pp. 465-578.
- [23] B.Shneiderman and R.Mayer, "Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results", Int'l J. Computer and Information Sci., 8,3 (June 1979), pp. 219-238.
- [24] B.Shneiderman, R.Mayer, D.McKay, and P.Heller, "Experimental Investigation of the Utility of Detailed Flowcharts in Programming", Comm.of the ACM 20,6 (June 1977), pp. 373-384.
- [25] N.F.Schneidewind and H.M.Hoffmann, "An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering SE-5, May 1979, pp. 276-286.
- [26] S.Sheppard, B.Curtis, M.Millman, and T.Love, "Modern Coding Practices and Programmer Performance", Computer 12,12, Dec. 1979, pp. 41-59.
- [26] M.Sime, A.Arblaster, and D.Guest, "Reducing Programming Errors in Nested Conditionals by Prescribing a Writing Procedure", International Journal of Man-Machine Studies 9, 1977, pp. 119-126.
- [28] M.Sime, T.Green, and D.Guest, "Psychological Evaluation of Two Conditional Constructions Used in Computer Languages", International Journal of Man-Machine Studies 5,1, 1973, pp. 105-113.
- [29] M.Sime, T.Green, and D.Guest, "Scope Marking in Computer Conditionals -- A Psychological Evaluation", International Journal of Man-Machine Studies 9,1, 1977, pp. 107-118.

- [30] G.Weinberg and E.Schulman, "Goals and Performance in Computer Programming", Human Factors 16,1, 1974, pp. 70-77.
- [31] L.Weissman, "Psychological Complexity of Computer Programs: An Experimental Methodology", SIGPLAN Notices 9,6 (June 1974), pp. 25-36.
- [32] E.Youngs, "Human Errors in Programming", International Journal of Man-Machine Studies 6, 1974, pp. 361-376.

DATA COLLECTION VALIDATION AND ANALYSIS

Victor R. Basili
Department of Computer Science
University of Maryland

One of the major problems with doing measurement of the software development process and the product is the ability to collect reliable data that can be used to understand and evaluate the development process and product and the various models and metrics. The data collection process consists of several phases--establishing the environment in which the project is being developed, the actual data collection process itself, the validation of the collection process and the data, and, finally, the careful analysis and interpretation of that data with respect to specific models and metrics. We will discuss each of these phases.

ESTABLISHING THE ENVIRONMENT

Before we begin collecting data, we must understand the various factors that affect software development. Data collection should begin with listing those factors one hopes to control, measure, and understand. In this way, we may characterize the environment, understand what we are studying, and be able to isolate the effects. One possible approach is to create categories of factors.

A partial list of factors is given below, categorized by their association with the problem, the people, the process, the product, the resources, and the tools. Some factors may fit in more than one category, but are listed only once.

1. People Factors - These include all the individuals involved in the software development process, including managers, analysts, designers, programmers, librarians, etc. People-related factors that can affect the development process include: number of people involved, level of expertise of the individual members, organization of the group, previous experience with the problem,

previous experience with the methodology, previous experience with working with other members of the group, ability to communicate, morale of the individuals, and capability of each individual.

2. Problem Factors - The problem is the application or task for which a software system is being developed. Problem-related factors include: type of problem (mathematical, database manipulation, etc.), relative newness to state-of-the-art requirements, magnitude of the problem susceptibility to change, new start or modification of an existing system, final product required, e.g., object code, source, documentation, etc., state of the problem definition, e.g., rough requirements vs. formal specification, importance of the problem, and constraints placed on the solution.

3. Process Factors - The process consists of the particular methodologies, techniques, and standards used in each area of the software development. Process factors include: Programming Languages, Process Design Language, Specification Language, Use of Librarian, Walk-throughs, Test Plan, Code Reading, Top Down Design, Top Down Development (stubs), Iterative Enhancement, Chief Programmer Team, Nassi-Shneiderman Charts, HIPO Charts, Data Flow Diagrams, Reporting Mechanisms, Structured Programming, and Milestones.

4. Product Factors - The product of a software development effort is the software system itself. Product factors include: deliverables, size in lines of code, words of memory, etc., efficiency tests, real-time requirements, correctness, portability, structure of control, in-line documentation, structure of data, number of modules, size of modules, connectivity of modules, target machine architecture, and overlay sizes.

5. Resource Factors - The resources are the nonhuman elements allocated and expanded to accomplish the software development. Resource factors include: target machine system, development machine system, development software,

deadlines, budget, and response times and turnaround times. (Note there is a relationship between resource and product factors in that the resources define a set of limits within which the product must perform. Sometimes these external constraints can be a dominating force on the product and sometimes they are only a minor factor, e.g., it is easy to get the product to perform well within the set of constraints.)

6. Tool Factors - The tools, although also a resource factor, are listed separately due to the important impact they have on development. Tools are the various supportive automated aids used during the various phases of the development process. Tool factors include: requirements analyzers, system design analyzers, source code analyzers (e.g., FACES), database systems, PDL processors, automatic flowcharters, automated development libraries, implementation languages, analysis facilities, testing tools, and maintenance tools.

COLLECTING THE DATA

Once it is clear what the environmental factors are, it is important that what data is needed be carefully considered. The data needed should be driven by the basic models and metrics that will be used and studied. However, since this may not always be known beforehand, especially in a research environment, we must also include a second level set of data that involves what we may want to know, model, or measure. Data collected in this bottom-up manner can be used to refine and modify the existing models and metrics and be used to better characterize our environment.

The actual collection process can take four basic formats: reporting forms, interviews, automatic collection using the computer system, and automated data analysis routines. The reporting forms are usually filled out by the various members of the development team from senior management down to clerical support.

The benefits of participants filling out the forms is that they can usually give detailed insights into what is really happening on the project and provide greater levels of detail in the data. Questions on a form can be much more specific than the kind of information one can collect automatically. On the other hand, automated data collection has the advantage of being more accurate since it is not as subject to human errors. It can also be done without the participants being aware of what specific activities and factors are being studied.

Form development is an art all by itself. First one needs to know what data is needed. This must be modified by what data the participants would be willing and able to answer accurately. One large factor here is sampling rate, that is, how often can the forms be filled out so that the participant is willing to do it and still remembers what it is you want to know. It is important that a certain amount of redundancy be built into the data collection process so that reliability checks can be made across the data forms.

Before forms are filled out, the participants should be given a training course in filling out the forms. They should be supplied with a glossary of terms, instructions on filling out the forms, and some sample filled out forms. It would also be helpful if the training session covered some of the models the data collectors had in mind so that the participants had a better idea of the kind of information that was wanted.

One representative set of forms (Basili, et al) may look as follows:

1. A General Project Summary - This form would be used to classify the project and will be used in conjunction with the other reporting forms to measure the estimated versus actual development progress. It should be filled out by the project manager at the beginning of the project, at each major milestone, and at the end. The final report should accurately describe the system development life cycle.

2. A Programmer/Analyst Survey - This form would classify the background of the personnel on each project. It should be filled out once at the start of the project by all personnel.

3. A Component Summary - This form would be used to keep track of the components of a system. A component is a piece of the system identified by name or common function (e.g., an entry in a tree chart or baseline diagram for the system at any point in time, or a shared section of data such as a COMMON clock). With the information on this form combined with the information on the Component Status Report, the structure and status of the system and its development can be monitored. This form is filled out for each component at the time that the component is defined, at the time it is completed, and at any point in time when a major modification is made. It should be filled out by the person responsible for that component.

4. A Component Status Report - This form would be used to keep track of the development of each component in the system. The form is turned in at the end of each week and for each component lists the number of hours spent on it. This form is filled out by persons working on the project.

5. A Resource Summary - This form keeps track of the project costs on a weekly basis. It is filled out by the project manager every week of the project duration. It should correlate closely with the component status report.

6. A Change Report Form - The change report form is filled out every time the system changes because of change or error in design, code, specifications or requirements. The form identifies the error, its cause and other facets of the project that are affected.

7. Computer Program Run Analysis - This form is used to monitor the computer activities used in the project. An entry is made every time the computer is used by the person initiating the run.

Interviews are used to validate the accuracy of the forms and to supplement the information contained on them in areas where it is impossible to expect reasonably accurate information in a form format. In the first case, spot check interviews are conducted with individuals filling out the forms to check that they have given correct information as interpreted by an independent observer. This would include agreement about such things as the cause of an error or at what point in the development process the error was caused or detected.

In the second case, interviews can be held to gather information in depth on several management decisions, e.g., why a particular personnel organization was chosen, why a particular set of people was picked, etc. These are the kinds of questions that often require discussion rather than a simple answer on a form.

The easiest and most accurate way to gather information is through an automated system. Throughout the history of the project, more and more emphasis should be placed on the automatic collection of data as we become more aware what data we want to collect, i.e., what data is the most valuable and what data we can or need to get, etc. More effort is required in the development or procurement of automatic collection tools.

The most basic information-gathering device is the program development library. The librarian can automatically record data and alleviate the clerical burden from the manager and the programmers. Copies of the current state of affairs of the development library can be periodically archived to preserve the history of the developing product.

A second technique for gathering data automatically is to analyze the product itself, gathering information about its structure using a program analyzer system. What data is gathered depends upon the particular product metrics.

The above data collected on the project should be stored in a computerized database. Data analysis routines can be written to collect derived data from the raw data in the database.

The data collection process is clearly iterative. The more we learn, the better informed we are about what other data we need and how better to collect it.

DATA VALIDATION

After archiving, the next stage is to validate the encoded data. The first step in the validation process is a review of the forms as they are handed in by someone connected with the data collection process to make sure all the forms have been handed in and that the appropriate fields have been filled out. The next step is to enter the data into the database through a program that checks the validity of the data format and rejects data out of the appropriate ranges. For example, this program can assure that all dates are legal dates and that system component names and programmer names are valid for the project by using a prestored list of component and programmer names.

Ideally, all data in the database should be reviewed by individuals who know what the data should look like. Clearly, this is expensive and not always possible. However, several projects should be reviewed for errors in detail and counts of the number of errors and types of errors kept so that error bounds can be calculated for the unchecked data. This allows data to be interpreted with the appropriate care.

Another type of validity check is to examine the consistency of the database by examining redundant data. This can be done by comparing similar data from different sources to assure the data is reasonably accurate. For example, if effort data is collected at the budget level (resource summary data) and at the individual programmer level (component status data), there should be a reasonable correlation between the two total efforts. Another approach is to use cluster

analysis to look for patterns of behavior that are indicative of errors in filling out the forms. For example, if all the change report forms filled out by a particular programmer fall into one cluster, it may imply that there is a bias in the data based upon the particular programmer.

It is clear data collection is a serious problem, especially in the collection of data on large programming projects across many environments where one set of forms may not be enough to capture what is happening in each of the environments. Unfortunately, if we are to compare projects, we do need common data and we need to know how valid that data is in each case so as not to draw improper conclusions.

DATA ANALYSIS AND RESULT REPORTING

After the environment has been established, the appropriate data collected and validated, the process of data analysis can begin. The first step entails fitting the data to the specific models and metrics and the interpretation of the results. If the data supports the model, then it reinforces our understanding of the software development process and product. If the data does not support the model, then we must further analyze the model and its application to the data and the data collection environment. It is possible that the data collection environment did not satisfy some of the assumptions of the model, explicit or otherwise. We can use this data to either refine or refute the model or to gain new insights into our software development environment. In any case, the application of the model to the data often generates more questions than it answers and sets the stage for new analysis and new data to be collected.

The data analysis process can be motivated by the different needs for understanding. When linked with various models and metrics, the data analysis can be used (1) to evaluate the software development process and product, (2) as a tool for software development, and (3) to monitor the stability and quality of

an existing product. The data collection and analysis process varies with each area of interest.

Better understanding of the software development process and the software development product is a critical need. Metrics can help in that understanding by allowing us to compare different products and different development environments and providing us with insights regarding their characteristics. Too often we think of all software as the same. Metrics can be used to delineate the various software products and environments.

Many metrics have as a major goal the evaluation of the quality of the process or product in a quality assurance environment. Thus a low score, on a metric like the number of errors, indicates something desirable about the quality of the process while a high score on the same metric indicates something quite undesirable about the product. Here data can be analyzed after the project is over.

A second use of metrics would be as a tool for development. In this case, the metric can act as feedback to the developer, letting him know how the development is progressing. It can be used to predict where the project is going by estimating future size or cost, or it may tell him his current design is too complicated and unstructured. Metrics should certainly be used across the entire life cycle and as early as possible to facilitate estimation as well as evaluation. Here data must be analyzed in real time and reports generated in a form easily understandable by the software developer.

A third use of metrics is to monitor the stability and quality of the product through maintenance and enhancement; that is, we can periodically recalculate a set of metrics to see if the product has changed character in some way. It can provide a much needed feedback during the maintenance period. If we find over a period of time that more and more control decisions have entered

the system, then something may have to be done to counteract this change in character. This last use of metrics is relativistic, requiring only a simple partial ordering to give us an indication of what is changed. A relative measure is clearly easier to validate than an absolute measure. The first two uses of metrics--the evaluation of the process and product and the tool of development--are predominantly absolute metrics; that is, there is nothing to compare them to within the same project. You may only compare their values with the values of the metrics on other projects. The drawback to an absolute metric is that we need some normalization and calibration factor to tell us what is good and what is bad. The data analysis environment here is somewhere between the two discussed above.

Data collected from any project must be interpreted with great care. One must know the nature of the project and its development environment. To use any model or metric, one must fully understand its assumptions as well as its strengths and weaknesses in order to interpret the results for the particular environment. One must generalize to other environments very cautiously and with great reserve. One unmeasured factor may account for a complete change in effect.

When reporting data, one should report the raw data, the various factors as they are understood, and, in the case of experiments, any statistical results independent of interpretation. It is important in reporting results to define the terms used as precisely as possible. There is a large communication problem due to imprecise units of measurement. For example, if size is reported in lines of source code, the measure is dependent upon the language used, whether or not comments are counted and the commenting convention, and whether or not only executable statements are counted. The difference in the figures could be of the order of two or three to one. Whenever the results of an analysis are

reported, it is important to publish error bounds, not just in the fit to the model, but in the actual data collection process itself.

There is a great deal of work to be done in the data collection process. More work must be done in defining terms. A variety of models must be developed which provide us with different viewpoints of the software development process and we must not fall into the trap of assuming that there is a single overall model of software and the software development process. Most important, because of the nature of experimentation and analysis and the many factors that contribute to software development, we must be ready to duplicate the studies and experiments of others and report our results in the open literature. It is only when a wealth of data is obtained to support a particular hypotheses that the software community will gain the confidence to believe in it.

Reference

(Basili, et al) Victor R. Basili, Marvin V. Zelkowitz, Frank E. McGarry, Robert W. Reiter, Walter F. Truszkowski, David L. Weiss, THE SOFTWARE ENGINEERING LABORATORY, Technical Report TR-535, May 1977, University of Maryland, Computer Science Center, College Park, Maryland 20742

EXPERIMENTAL EVALUATION OF SOFTWARE CHARACTERISTICS AND PROGRAMMER PERFORMANCE

Bill Curtis
Information Systems Programs
General Electric Company
Arlington, Virginia

Cause-Effect Relationships

Most studies on software metrics do not demonstrate cause-effect relationships between software characteristics and programmer performance. That is, there were uncontrolled factors in the data collection environment which could have influenced the observed data. These alternate explanations of the results dilute any statement of cause and effect. Although structural equation techniques (Duncan, 1975; Heise, 1975) allow an investigation of whether the data are consistent with one or more theoretical models, a causal test of theory will require a rigorously controlled experiment. According to Cattell (1966):

An experiment is a recording of observations. . .made by defined and recorded operations and in defined conditions followed by examination of the data...for the existence of significant relations. (p. 20)

Two important characteristics of an experiment are that its data collection procedures are repeatable and that each experimental event result in only one from among a defined set of possible outcomes (Hays, 1963). An experiment does not prove an hypothesis. It does, however, allow for the rejection of competing alternative explanations of a phenomenon.

The confidence which can be placed in a cause-effect statement is determined by the control over extraneous variables exercised in the collection of data. For instance, Milliman and Curtis (1980) reported a field study in which a software development project guided by modern programming practices produced higher quality code with less effort and experienced fewer system test errors when compared to a sister project in the same environment which did not observe these practices. Although many of the environmental factors were controlled, an alternate explanation of the results was that the project guided by modern practices was performed by a programming team with more capable personnel.

An important characteristic of the classical experimental method in behavioral science is the random assignment of participants to conditions (Fisher, 1935). By removing any systematic variation in the ability, motivation, etc. of participants across experimental

conditions, this method supposedly eliminates the hypothesis that experimental effects are due to individual differences among participants. Assigning a morning class to one condition and an afternoon class to another condition does not constitute random assignment, since students rarely choose class times on a random basis. However, if classes are the unit of study, the problem can be solved by randomly assigning a number of classes to each experimental condition. Random assignment has been a problem in testing causal relationships in field studies on actual software development projects.

There is often a conflict between what Campbell and Stanley (1966) describe as the internal and external validity of an experiment. Internal validity concerns the rigor with which experimental controls are able to eliminate alternate explanations of the data. External validity concerns the degree to which the experimental situation resembles typical conditions surrounding the phenomena under study. Thus, internal validity expresses the degree of faith in causal explanations, while external validity describes the generalizability of the results to actual situations.

In software engineering research, rigorous experimental controls are difficult to achieve on software projects and laboratory studies often seem contrived. External validity is probably a greater problem in studying process factors such as the organization of programming teams than in studying product factors such as software characteristics. That is, the environmental conditions surrounding software development which are difficult to replicate in the laboratory would probably have a greater effect on the functioning of programming teams than on a programmer's comprehension of code.

Reviews of the experimental research in software engineering have been compiled by Atwood, Ramsey, Hooper, and Kullas (1979), Moher and Schneider (1979), and Shneiderman (1980). Topics which have been submitted to experimental evaluation include batch versus interactive programming, programming style factors (e.g., indented listings, mnemonic variable names, and commenting), control structures, documentation formats, code review techniques, and programmer team organization. In the next section I will review one of the more extensive areas of experimental research in software engineering: the evaluation of conditional statements and control flow. This topic was not chosen because it was believed to be more important than other subjects. Rather, it was chosen because several programs of research have investigated this topic and because the conditional statement has been a focus of argument since it was originally assailed by Dijkstra in 1968. Control statements have been a concern of the structured programming movement, and the results reported here evaluate their most effective implementation.

Conditional Statements and Control Flow

Sime, Green, and their colleagues at Sheffield University have been studying the difficulty people experience in working with conditional statements. In their first experiment Sime, Green, and Guest (1973) compared the ability of non-programmers to develop a simple algorithm with either nested or branch-to-label conditionals. Nesting implies the embedding of a conditional statement within one of the branches of another conditional. Nested structures are designed to make this embedding more visible and comprehensible to a programmer. Branch-to-label structures obscure the visibility of embedded conditions, since the "true" branch of a conditional statement sends the control elsewhere in the program to a statement with a specified label.

The conditional for the nested language was an IF-THEN-OTHERWISE construct similar to conditionals used in Algol, PL/I, and Pascal. This conditional construct is written:

```
IF [condition] THEN [process 1]
  OTHERWISE [process 2]
```

The branch-to-label conditional was the IF-GOTO construct of Fortran and Basic which Dijkstra (1968) considered harmful. This conditional is written:

```
IF [condition] GOTO L1
  [process 2]
L1 [process 1]
```

Participants used one of these micro-languages to build an algorithm which organized a set of cooking instructions depending on the attributes of the vegetable to be cooked. Sime et al. found that participants using the GOTO construct finished fewer problems, took longer to complete them, and made more semantic (e.g., logic) errors in building their algorithms than participants using the IF-THEN-OTHERWISE construct.

In a second experiment, Sime, Green, and Guest (1977) investigated different techniques for marking the scope of the processes subsumed under each branch of a conditional statement. In addition to the IF-GOTO conditional, they defined a nested BEGIN-END and a nested IF-NOT-END representing two different structures for marking the scope of each branch in nested conditionals. The BEGIN and END statements mark the scope of processes performed under one branch of an IF-THEN-OTHERWISE construct, while the IF-NOT-END uses a more redundant scope marker by repeating the condition whose truth is being tested. The IF-NOT-END construct is written:

```
IF (condition) (process 1)
  NOT (condition) (process 2)
  END (condition)
```


Sime et al. (1977) found that more semantic (algorithmic) errors occurred in the IF-GOTO language, while errors in the nested languages were primarily syntactic (grammatical). The BEGIN-END construct produced more syntactic errors and only half as many successful first runs as the other constructs. Errors were debugged ten times faster in the IF-NOT-END condition, which proved to be the most error free construct.

Based on the results of this second experiment, Sime, et al. (1972) proposed that information is easier to extract from some languages than others. They distinguished two types of information: sequence and taxon. Sequence information involves establishing or tracing the flow of control and events forward through a program. Taxon information involves the hierarchical arrangement of conditions and processes within a program. Such information is important when tracing backward through a program to determine what conditions must be satisfied for a process to be executed. Sime et al. hypothesized that sequence information is more easily obtained from a nested language, while taxon information is more easily extracted from a nested language which also contains the redundant conditional expressions. In two subsequent studies Green (1977) validated the hypothesis about differences between sequence and taxon information in research with professional programmers.

It is important to recognize that program comprehension is not a unidimensional cognitive process. Rather, different types of human information processing are required by different types of software tasks. Green demonstrated that certain constructs were more helpful for performing certain software tasks. Software engineering techniques may differ in the benefits they offer to different programming tasks, since they differ in the types of human information processing that they assist. Models of programmer performance need to take this interaction between task and process into account if valid hypotheses are to be derived from them.

Since the IF-NOT-END construct is not implemented in existing languages, Sime, Arblaster, and Green (1977) investigated ways to improve the use of the BEGIN-END conditional markers for IF-THEN-OTHERWISE constructs. They developed a tool which would automatically build the syntactic portions of a conditional statement once the user chose the expression to be tested. In a second experimental condition, they developed an explicit writing procedure for helping participants develop the syntactic elements of a conditional statement. This procedure involved writing the syntax of the outermost conditional first, and then writing the syntax

of conditionals nested within it. In the final condition participants were left to their own ways of creating the conditional constructs.

Sime et al. found that participants solved more problems correctly on their first attempt using the automated tool, but that a writing procedure was almost as effective. The writing procedure reduced the number of syntactic errors, which had been the major problem with the BEGIN-END construct in earlier studies. Syntactic errors were not possible with the automated tool. The writing procedure and automated tools helped participants dispense with syntactic considerations quickly, so that they could spend more time concentrating on the semantic portion of the program (i.e., the function which was to be performed). However, once an error was made, it was equally difficult to correct regardless of the condition. Thus, writing procedures and tools primarily increased the accuracy of the initial implementation.

The extensive program of research by Sime, Green, and their colleagues demonstrated:

- the superiority of nested over branch-to-label conditionals;
- the advantage of redundant expression of controlling conditions at the entrance to each conditional branch;
- that the benefits of a software practice may vary with the nature of the task; and
- that a standard procedure for generating the syntax of a conditional statement can improve coding speed and accuracy.

Overall, these results indicate that the more visible and predictable the control flow of a program, the easier it is to work with.

In a separate attempt to evaluate the GOTO statement, Lucas and Kaplan (1974) instructed 32 students to develop a file update program in PL/C, and half were further instructed to avoid the use of GOTOs. However, programmers in the GOTO-less condition were not trained in using alternate conditional constructs. Not surprisingly, the GOTO-less group required more runs to develop their programs. In a subsequent task, all participants were required to make a specified modification to a structured program. Contrary to results on the earlier task, the group which had earlier struggled to write GOTO-less code made quicker modifications which required less compile time and storage space.

Weissman (1974) also investigated the comprehension of PL/I programs written in versions whose control flow was either 1) structured, 2) unstructured but simple, or 3) unstructured and complex. Participants were given comprehension quizzes and required to make modifications to the programs. Higher performance scores were typically obtained on the structured rather than unstructured versions, and participants reported feeling more comfortable with structured code. Love (1977) subsequently found that graduate students could comprehend programs with a simplified control flow more easily than programs with a more complex control flow.

Recently, a series of experiments evaluating the benefits of structured code for professional programmers was reported by Sheppard, Curtis, Milliman, and Love (1979). In the first two experiments three versions of control flow performing identical functions were defined for each of several Fortran programs. One version was structured to be consistent with the principles of structured coding described by Dijkstra (1972). Because structured constructs are sometimes awkward to implement in Fortran IV (Tenny, 1974), a more naturally structured control flow was devised which allowed limited deviations from strict structuring: multiple returns, judicious backward GOTOs, and forward mid-loop exits from a DO. Finally, a deliberately convoluted version was developed which included constructs that had not been permitted in the structured or naturally structured versions.

The first experiment was on comprehension and required participants to reconstruct a previously studied program. As expected, the convoluted control flow was significantly more difficult to comprehend than naturally structured programs. Differences between naturally and strictly structured programs were insignificant.

In a second experiment Sheppard et al. instructed programmers to make specified modifications to three programs, each of which was written in the three versions of control flow described previously. A significantly higher percentage of the steps required to complete each modification was correctly implemented in the strictly structured programs when compared to convoluted ones. No statistically significant differences appeared between the two versions of structured control flow.

Results from the first two experiments suggested that the presence of a consistent structured discipline in the code was beneficial, and minor deviations from strict structuring did not adversely affect performance. This premise was tested in a third experiment which compared the two versions of structured Fortran IV to Fortran 77. Fortran 77 contains the IF-THEN-ELSE, DO-WHILE, and DO-UNTIL con-

structs usually associated with structured coding. They measured how long a programmer took to find a simple error embedded in a program. No differences were attributable to the type of structured control flow, replicating similar results in the first two experiments. The advantage of structured coding appears to reside in the ability of the programmer to develop expectations about the flow of control - expectations which are not seriously violated by minor deviations from strict structuring.

The research reviewed here indicates that programs in which some form of structured coding is enforced will be easier to comprehend and modify than programs in which such coding discipline is not enforced. It is not clear that structured coding will improve the productivity of programmers during implementation. However, some productivity improvements may be observed if less severe, more easily corrected errors are made using structured constructs (as suggested in the data of Sime and his colleagues). Structured coding should reduce the costs of maintenance since such programs are less psychologically complex to repair and modify. Experiments such as these can provide valuable guidance for decisions about an optimal mix of software standards and practices.

Problems in Experimental Research

It is important to recognize the benefits and limitations of controlled laboratory research. On the positive side, rigorous controls allow experimenters to isolate the effects of experimentally manipulated factors and identify possible cause-effect relationships in the data. On the other hand, the limitations of controlled research restrict the generalizations which can be made from the data. Laboratory research has an air of artificiality, regardless of how realistic researchers make the tasks.

Several problems attendant to most current empirical validation studies severely limit the generalizability of conclusions which can be drawn from them (Brooks, 1980). For instance, program sizes have frequently been restricted because of limitations in the research situation. This problem is characteristic of experimental research where time limitations do not allow participants to perform experimental tasks involving the coding or design of large systems. Also, since new factors come into play in the development of large systems (e.g., team interactions), the magnitude of a technique's effect on project performance may differ markedly from its effect in the laboratory.

The nature of the applications studied are often limited by the environments from which the programs are drawn (e.g., military versus commercial systems, real-time versus

non-real-time systems, etc.). Further, there is frequently little assessment of whether results will hold up across programming languages. It is extremely difficult to perform evaluative research over a broad range of applications, especially when experimental procedures are used. Thus, empirical results should be replicated over a series of studies on different types of programs and in languages other than Fortran.

Another problem arises with what Sackman, Erickson, and Grant (1968) and Sheppard et al. (1979) observed to be 25 or 30 to 1 differences in performance among programmers. This dramatic variation in performance scores can easily disguise relationships between software characteristics and associated criteria. That is, differences in the time or accuracy of performing some software task can often be attributed more easily to differences among programmers than to differences in software characteristics. Careful attention to experimental design is required to control this problem.

If generalizations are to be made about the performance of professional programmers, this is the population that should be studied rather than novices. As is true in most fields, there are qualitative differences in the problem-solving processes of experts and novices (Simon, 1979). However, the advantage of some techniques is the ease with which they are learned, and novices are the appropriate population for studying such benefits. Attempts to generalize experimental results must also be tempered by an understanding of how real-world factors affect outcomes. Data should be collected in actual programming environments to both validate conclusions drawn from the laboratory and determine the influence of real-world factors.

Measurement and experimentation are complementary processes. The results of an experiment can be no more valid than the measurement of the constructs investigated. The development of sound measurement techniques is a prerequisite of good experimentation. Many studies have elaborately defined the independent variables (e.g., the software practice to be varied) and hastily employed a handy but poorly developed dependent measure (criterion). For instance, program reconstruction is not a good measure of comprehension, since it is affected by memorization skills. Results from experiments with inadequate dependent variables are difficult to explain because they are confounded with other processes.

Results are far more impressive when they emerge from a program of research rather than from one-shot studies. Programs of research benefit from several advantages, one of the most important being the opportunity to replicate findings. When a basic finding (e.g., the benefit of structured coding) can be replicated over several different tasks

(comprehension, modification, etc.) it becomes much more convincing. A series of studies also result in deeper explication of both the important factors governing a process and the limits of their effects. Performing a series of studies also affords an opportunity to improve measurement and experimental methods. Thus, the reliability and validity of results can be improved in succeeding studies.

Approaches to Experimental Research

There are three primary experimental approaches appropriate for research on problems in software engineering: the case study, the classical multi-participant experiment, and simulation. Each of these approaches has advantages for studying certain types of problems. The paper by Fred Sayward in this collection reviews the case study approach, while I have concentrated on the classical multi-participant experiment. This latter technique is primarily a method of determining cause-effect relationships for a limited number of factors.

The case study approach can be used in either single-subject research or project field studies. In single-subject research the case study involves collecting continuous protocol data from a participant who is performing a software task, most frequently the design and implementation of an algorithm. A critical problem comes in defining a unit of behavior and in avoiding the many sources which contaminate protocol data. Such data is often difficult to interpret, but holds promise for modeling the human cognitive processes involved in software development. These models are more difficult to build from the results of classical multi-participant experiments, although protocol data can be collected in such experiments. Thus, the single-subject approach may be more appropriate for initial data gathering from which models can be built and experimentally tested using other approaches. In field studies of software development projects, longitudinal data can be collected which help describe actual software development processes and determine the effects of environmental factors not studied in the laboratory (Milliman & Curtis, 1980). Such data are appropriate for developing the resource models described in the paper by Basili in this collection.

The experimental investigation of software design techniques for large systems would be difficult or impossible by any of the methods described previously. That is, funding is not available for implementing a quarter-million line software system in parallel by two or more project teams, each using a different design technique (e.g., data structured versus functionally structured). However, simulation tools are becoming available which will allow us to simulate the full implementation of a software design.

An experiment on design techniques using a simulation tool need go no further than the initial design specification in order to obtain worthwhile results. System size, performance, and some quality metrics would be generated in such an experiment. Simulation techniques hold great promise for many questions in software engineering which seem intractable to current experimental methodologies.

Conclusion

There is a steadily growing body of experimental research on software engineering techniques and their effects on programmer productivity. These experiments have been performed on problems which have been the easiest to operationally define and control in an experimental setting. Such problems involve the use of structured coding, flow-charts, indented listings, etc. Now that results are beginning to emerge on these topics, evaluative data are desperately needed on the most effective tools for assisting programmers in software development and maintenance.

The greatest opportunities for control over the outcomes of a software project are those gained during the development of requirements and the initial design specification. Almost no experimental research has evaluated various methods for creating and recording requirements and initial design specifications. Lance Miller and his colleagues at IBM's Watson Research Center, however, have performed some research on problem-solving techniques which relate to software development.

Another critical area of research involves language differences. Experiments such as those reported here on structured coding and those on strongly versus weakly typed languages performed by Gannon (1977) have provided some initial results on language characteristics. However, larger experiments comparing the usefulness of various types of languages (e.g., APL vs. Ada vs. Fortran) for implementing different types of algorithms need to be performed. The advent of non-procedural languages invites such comparisons. New methodologies for performing such experiments need to be developed, and some may be an order of magnitude more difficult to implement than the methodologies currently in use. However, simulation techniques may provide a new source of data which avoids problems associated with individual differences among programmers. Such experiments will provide critical data on the most important factors affecting programmer productivity and software quality.

ACKNOWLEDGEMENTS

I would like to thank Laszlo Belady and Sylvia Shepard, and Drs. Elizabeth Kruesi and John O'Hare for their thoughts and comments. Portions of this paper were drawn from work supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N00014-79-C-0595). However, the opinions expressed in this paper are not necessarily those of the Department of the Navy. Reprints can be obtained from Dr. Bill Curtis; General Electric Company, Suite 200; 1755 Jefferson Davis Highway; Arlington, VA 22202.

REFERENCES

- Atwood, M.E., Ramsey, H.R., Hooper, J.N., & Kullas, D.A. Annotated Bibliography on Human Factors in Software Development (Tech. Report TR-P-79-1). Alexandria, VA: Army Research Institute, 1979. (NTIS No. AD A071 113).
- Brooks, R. Studying programmer behavior experimentally: The problems of proper methodology. Communications of the ACM, 1980, 23, 207-213.
- Campbell, D. & Stanley, J. C. Experimental and quasi-Experimental Designs for Research. Chicago: Rand-McNally, 1966.
- Cattell, R. B. The principles of experimental design and analysis in relation to theory building. In R.B. Cattell (Ed.), Handbook of Multivariate Experimental Psychology. Chicago: Rand-McNally, 1966, 19-66.
- Dijkstra, E. W. GO TO statement considered harmful. Communications of the ACM, 1968, 11, 147-148.
- Dijkstra, E. W. Notes on structured programming. In O. J. Dahl, E. W. Dijkstra, & C.A.R. Hoare (Eds.) Structured Programming. New York: Academic, 1972, 1-82.
- Duncan, O. D. Introduction to Structural Equation Models. New York: Academic, 1975.
- Fisher, R. A. The Design of Experiments. London: Oliver & Boyd, 1935.
- Gannon, J. G. An experimental evaluation of data type conventions. Communications of the ACM, 1977, 20, 584-595.
- Green, T. R. G. Conditional program statements and their comprehensibility to professional programmers. Journal of Occupational Psychology, 1977, 50, 93-109.
- Hays, W. L. Statistics. New York: Holt, Rinehart, & Winston, 1973.
- Heise, D. R. Causal Analysis. New York: Wiley, 1975.
- Love, T. An experimental investigation of the effect of program structure on program understanding. SIGPLAN Notices, 1977, 12 (3), 105-113.

Lucas, H. C. & Kaplan, R. B. A structured programming experiment. The Computer Journal, 1974, 19, 136-138.

Milliman, P. & Curtis, B. A Matched Project Evaluation of Modern Programming Practices (RADC-TR-80-6, 2 vols.). Griffiss AFB, NY: Rome Air Development Center, 1980.

Moher, T. & Schneider, G. M. Methods for Improving Controlled Experimentation in Software Engineering (Tech. Rep. 80-8). Minneapolis, MN: University of Minnesota, Computer Science Department, 1980.

Sackman, H., Erickson, W. J., & Grant, E. E. Exploratory and experimental studies comparing on-line and off-line programming performance. Communications of the ACM, 1968, 11, 3-11.

Sheppard, S. B., Curtis, B., Milliman, P., & Love, T. Modern coding practices and programmer performance. Computer, 1979, 12, 41-49.

Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, MA: Winthrop, 1980.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. Reducing programming errors in nested conditionals by prescribing a writing procedure. International Journal of Man-Machine Studies, 1977, 9, 119-126.

Sime, M. E., Green, T. R. G., & Guest, D. J. Psychological evaluation of two conditional constructions used in computer languages. International Journal of Man-Machine Studies, 1973, 5, 105-113.

Sime, M. E., Green, T. R. G., & Guest, D. J. Scope marking in computer conditionals - A psychological evaluation. International Journal of Man-Machine Studies, 1977, 9, 107-118.

Simon, H. A. Information processing models of cognition. In M. R. Rosenzweig & L. W. Porter (Eds.) Annual Review of Psychology (Vol. 30). Palo Alto, CA: Annual Reviews, 1979, 363-396.

Tenny, T. Structured programming in Fortran. Datamation, 1974, 20 (7), 110-115.

Weissman, L. M. A Method for Studying the Psychological Complexity of Computer Programs (Tech. Rep. TR-CSRG 37). Toronto: University of Toronto, Department of Computer Science, 1974.

SOFTWARE PROJECT FORECASTING

Richard A. DeMillo
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 320332

Richard J. Lipton
Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08750

THIS IS A DRAFT COPY OF A PAPER PREPARED FOR THE JUNE 30
ONR MEETING ON SOFTWARE METRICS -- SINCE THIS IS ONLY A
PRELIMINARY VERSION PLEASE DO NOT CIRCULATE OR CITE -
THIS PAPER WITHOUT CONSULTING FIRST WITH THE AUTHORS

INTRODUCTION

A characterizing feature of the sciences is that they seek to explain, describe and predict phenomena. While there are varying degrees of exactness in the sciences (the predictions of physics are of a quantitative character, different than, say, the predictions of economics) the basis of scientific activity is rational and objective. This is what distinguishes economics from astrology -- even though it might be argued that economic and astrological predictions are equally vague, economic predictions are the result of rational analysis of evidence. Thus we intend to divide sciences from non-sciences on the basis of the rational nature of the activities; for a detailed discussion, see [1].

Let's now look at the problem of "measuring" software.

It is evident from the remaining papers in this collection that aside from a well-founded concern over methodological issues, the principle aim of studying software metrics is not the static determination of software properties, but is rather the scientific prediction of phenomena during the software lifecycle. Indeed, Perlis, Sayward and Shaw point out (cf. [2]): "The purpose of software metrics is to provide aids for making the optimal choice... at several points in the life cycle." They go on to illustrate the nature of the decision points.

- How long will it take to produce the software?
- When will it have to be replaced?
- What are the manpower requirements?
- How will the availability of tool X affect factor Y?
- How close to its resource limits will the system run?
- Will it run reliably?

These are manifestly problems of prediction. As intriguing as the issues of explanation and description may be -- our ability to usefully model software is still primitive -- the exigencies of governmental and commercial computing demand a reasonable facility in forecasting critical software project parameters. We will address this forecasting problem in the sequel. We pretend no well-formed answers here. In fact, our goal is the rather modest one of pointing out that there is a scientific (although possibly inexact) component of the problem that is not adequately conveyed by the term

"software metrics": the use of past information to predict the future of a software system.

ANALOGIES

Inevitably, primitive sciences are compared to physics, the reliable standard of scientific rigor and success. Of course, inexact sciences fare very badly in the comparison, but often the reason they fare so badly is not understood. Physics stands almost alone among the sciences in the exactness and simplicity of its theories. The price paid is the complexity of the situations that can be profitably handled by the theories. Predicting the behavior of complex systems -- particularly those involving human interactions -- is almost never carried out by deducing from first principles, that is, from physical theory. Tim Standish [3] points out that most scientific knowledge is organized so that phenomena at one level can be explained in terms of (or "reduced to") phenomena at a more basic level; for example, physical chemistry explains chemical behavior in purely physical terms. Only rarely, however, is it possible to compose several such reductions in intellectually manageable fashion. Thus while it is possible to imagine physical explanations of biological phenomena, biological explanations of psychological phenomena, and a psychological basis for social behavior, it is extremely unlikely that there will ever be a physical theory of social behavior.

Jim Browne is correct: "There are analogies from other sciences." [4]. We even agree that the fundamental issues

are predictive and phenomenological. We differ in the choice of analogies, and we also differ on the role that measurement plays in constructing useful forecasts. It seems very unlikely that the theory of software forecasting is like physical theory at an early stage in its development. We have argued elsewhere [5] that software exhibits much complexity and ad-hocery, features that cannot easily be abstracted from real situations or simplified with approximations. The prediction problem for software is more akin to the corresponding problems in those disciplines that deal with complex systems; rather than turn to physics for our methodology, we should turn to the less exact sciences -- fields such as meteorology, economics, political science and even the construction industry in which the explanatory component is subjugated to the predictive component primarily because of the extreme public and social significance of the predictions.

Why distinguish at all between explanatory and predictive theories? After all, predictive and explanatory assertions are logically equivalent. They both use evidence to convince a listener of an hypothesis. They may both give "laws" concerning an effect X; the exactness of the laws may vary from that of the quasi-laws [1] (X is asserted to be present except in certain exceptional cases) to the more exact statistical laws (X is asserted to be present in a stated fraction of the observations) to the mathematically exact physical laws. Predictions and explanations are, however, distinguished in a fundamental way: in explanatory

theories the hypotheses concern events which are past, while predictions are hypotheses concerning future events. In logical terms, for an explanation to establish its conclusion, it must be more credible than its negation. On the other hand, a prediction must only be rendered more tenable than the alternatives!

METEOROLOGY

The physical basis of meteorological theory fits comfortably on a moderately large blackboard. It consists primarily of six equations of fluid dynamics which locally predict the state of the atmosphere from the current state:

$$\text{STATE}(\text{new}) = f(\text{STATE}(\text{old})).$$

By observing pressure, temperature and other meteorologically interesting variables (in fact, only six independent variables and a few thermodynamical constants are involved) and calculating their rate of change, the state equations allow extrapolation over short time periods to new, or predicted, values (see the accounts in [5] or [7]).

It is amusing that the first attempts at a meteorological theory were directed toward an explanatory theory of the weather; this was attempted by ancient Greek philosophers. But predicting the weather in the vicinity

the Mediterranean Sea is not a very pressing concern. To Europeans a thousand years hence, however, the state of the weather was a subject of intense interest -- crops, harvests of fish, and trade routes depended on the vagaries of the less temperate climate. It is in the transition from Aristotle's speculation concerning the nature of the winds to the modern large-scale calculations that give rise to daily forecasts that there is a lesson for software forecasting.

The first stage in predicting weather most likely took the form of attempting to codify the "portents" of change: "They...were not so much concerned with explaining why weather happened as they were in predicting it, and [the early Europeans] gradually built up a huge folklore and literature of portents -- the unseasonable migration of birds, hibernation of wild beasts, unusual sexual behavior of farm animals, color of the sunset,..." ([6], p. 123). Such understanding is entirely qualitative, and although it is tempting to try, building a quantitative, predictive theory by improving on the portents is not productive. What was needed in meteorology was, first, a proper concept of primary data. Not until the middle of the seventeenth century was there a means to observe atmospheric temperature and pressure. Measurements by themselves told little about climate beyond what was obviously revealed directly by the observation (e.g., "It's cold out there!") It was after Newton (and later Bernoulli, Euler and Boyle) that a coherent fluid mechanics began to emerge.

It took nearly two hundred years to get from Bernoulli to the six equations of state for the atmosphere. These equations are, however, analytically intractable. The only hope of obtaining high quality quantitative predictions of the weather lay in massive computation. A proposal made by Richardson in 1922 involved the hand calculation of such nonlinear systems of equations by numerical techniques that are close to what is used today [7]. Without computers, however, Richardson could only speculate on the actual mechanism of carrying out the necessary calculations:

[Richardson] describes a phantasmagorical vision of the "weather factory" -- a huge organization of specialized human computers, housed in something like Albert Hall, directed by a mathematical conductor perched on a raised pulpit, and communicating by telegraph, flashing colored lights, and pneumatic conveyer tubes...In this fantasy, he estimated that, even using the newfangled desk calculators, it would take about 64,000 human automata just to predict the weather as fast as it actually happens in nature. Richardson's preface concludes with a rather wistful but prophetic statement: 'Perhaps someday in the dim future it will be possible to advance the computations faster than the weather advances and at a cost less than the saving to mankind due to the information gained. But that is a dream' ([5], p. 138).

The nearly simultaneous advance of sophisticated numerical analytical techniques and high speed digital computation allowed the fulfillment of Richardson's dream in essentially its original form.

The characterizing features of modern meteorological forecasting are that, first, extensive primary data is

gathered, second, accurate microscopic theories of atmospheric behavior are available, and, third the microscopic prediction -- obtained from the local data -- are pieced together using massive computation.

A DIGRESSION ON MEASUREMENT

Since measurement of atmospheric pressure and temperature enter into our meteorological analogy, we should digress for a moment to consider the notion of measurement of software. The remaining papers in this collection refer to software "metrics". The term metrics refers to "indices of merit that can support quantitative comparisons and evaluations..." [3]. In the context of predictive modeling (that is of predicting the future from the past), it is more convenient to think in terms of observable software properties -- particularly those that can be numerically characterized and objectively recorded -- in other words, to think in terms of measurements instead of metrics. The distinction is not totally pedantic. There is a rich theory of measurement that guides the development of other models [9,10,11,12], and most importantly can be used to insure that there is a precise sense in which hypotheses that are formulated about the measured quantities are meaningful!

By a measurement is meant the assignment of numbers to represent properties of material systems; since by a system we mean a collection of objects or events, the properties of

the system are given by relations between the objects/events. For reasons of intellectual economy a scientist usually isolates one aspect of the system to study; that is, he focuses on one relational system. So, a measurement -- an assignment of numerals to objects or events according to certain rules [9] -- can be defined to be a mapping f from a relational system (A,R) , where A is a set and R is a binary relation defined on A , to a set of numbers. Since the numbers "represent" the relation, we should insist that:

$$aRb \text{ iff } f(a) > f(b),$$

whenever a and b are objects in A . More concisely, measurements are defined to be homomorphisms that preserve certain basic relations.

Thus, a basic measurement of, say, temperature is obtained by the assignment of a number by a well-defined rule (e.g., the height of fluid in a standard thermometer). This homomorphism is not uniquely defined, however. It is possible -- and common -- to define differing scales for measuring temperature. The scientifically meaningful statements that can be made about temperature do not depend on the scale; that is, they remain valid under rescaling. Just which rescalings are allowed depends on the properties of the relational system (A,R) . Similarly, scale types can be characterized by the admissible rescalings as summarized in Figure 1 (see [10]).

ADMISSIBLE TRANSFORMATIONS	SCALE TYPE	EXAMPLE
$\delta(x) = x$	absolute	census counting
$\delta(x) = ax$ $a > 0$	ratio	time interval length
$\delta(x) = ax + b$ $a > 0$	interval	time, temperature
$x \geq y$ implies $\delta(x) \geq \delta(y)$	ordinal	preference
δ is 1-1	nominal	labels

Figure 1. Common Rescalings

Strictly speaking, an empirical statement is meaningful provided its truth is invariant under an admissible transformation. We can safely assert that 100 degrees C is the boiling point of water, since the statement is true under the rescaling $a = 9/5$ and $b = 32$. It makes no sense to assert that the temperature on March 15 is twice the temperature on November 4, since temperature is not defined on a ratio scale -- the ratio of temperatures depends on the scale and is therefore not invariant under the rescaling.

As a warm-up, let's look at some empirical statements about software.

- I. The length of Program A is at least 100.
- II. Program A is 100 lines long.
- III. Program B took 3 months to write.

- IV. Program C is twice as long as Program D
- V. Program C is 50 lines longer than Program D.
- VI. The cost of maintaining Program E is twice that of maintaining Program F.
- VII. Program F is twice as maintainable as Program E.

Statement I does not make reference to a particular scale, so it does not make sense, whereas Statement II does make sense. Similarly, Statement III is a perfectly reasonable factual statement. If the expected scales are provided for Statements IV and VI, they are meaningful, but as written they are technically meaningless. Statement V, however refers to an ratio scale, on which intervals make sense. Finally Statement VII forms a ratio on an ordinal scale, which is meaningless.

From the standpoint of measurement theory, many of the derived measurements of software that have been proposed [13] are meaningless.

Example 1. Programming Effort Equation

The total effort E in number of man months is

$$E = 2.7v + 12lw + 26x + 12y + 22z - 497.$$

The interpretation of the variables and their scales types are given in the following table.

VARIABLE	INTERPRETATION	SCALE
v	number of instructions	ratio
w	subjective complexity	ordinal
x	no. external documents	absolute
y	no. internal documents	absolute
z	size in words	ratio

This example illustrates a common shortcoming of current attempts at fundamental and derived measurements. Not only is the equation dimensionally inconsistent (no. of documents + no. of instructions + words + complexity \neq man-months), it does not rescale: the truth of the equation cannot be invariant under the required transformations.

Example 2. Another Programming Effort Equation

The total effort E in man-months is

$$E = 5.2(L^{.91}).$$

In this equation, L is the program size in thousands of lines of code, so that both E and L are expressed in a ratio scale. But the measurement is not invariant under the transformation $L \rightarrow aL$ and so is meaningless.

Example 3. Life Cycle Cost

A basic measurement that does satisfy the requirements of measurement theory is the equation for life cycle cost in dollars LC , expressed as a function of the cost in man years, M ,

AD-A087 412

YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE

F/G 9/2

DRAFT SOFTWARE METRICS PANELS FINAL REPORT. PAPERS PRESENTED AT--ETC(U)

JUN 80 A J PERLIS, F G SAYWARD, M SHAW

N00014-79-C-0672

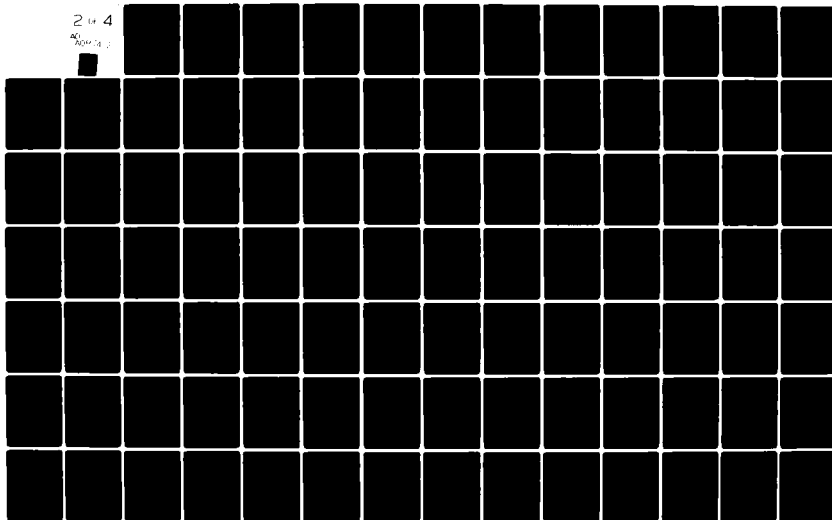
UNCLASSIFIED

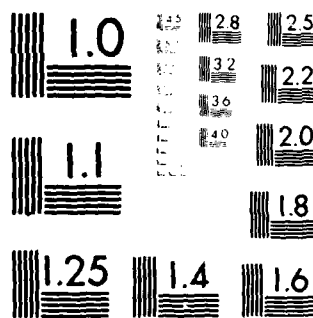
RR-182/80

NL

2 of 4

AD-A087 412





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

and the average cost per man year, C:

$$L = MC.$$

Example 4. Error Seeding

The technique of introducing artificial errors into a program, testing the program and determining the ratio of seeded to natural errors can be used to estimate the number of initial errors in a program, by the equation

$$N = ST/C,$$

where N is the estimate of the initial number of errors, S is the total number of errors sampled, T is the number of "tagged" or seeded errors and C is the number of errors in common in the counts S, T. Since the only admissible transformation is the identity, the equation is technically meaningful.

As a guide to fundamental measurement in software forecasting measurement theory suggests a more thorough study of the underlying relations to be measured. In particular, if the underlying mechanisms are to be exposed, the most basic methodological analyses suggest that it is prudent to at least determine the scale type first. At least that leads the investigator to propose and experiment on meaningful hypotheses.

REALISTIC FORECASTING GOALS

Examples 1 and 2 of the previous section illustrate a good deal of what is wrong with current approaches to software metrics. Not only do the equations suffer from the technical defects cited above, they are also unlikely candidates for useful laws: they are too simple! The number and quality of interactions that must take place to produce a software system mitigate against a forecasting problem that can be easily solved on a hand calculator. The forecasting models that are the most realistic are also the most demanding in terms of computation and data gathering. For example, the controversial world dynamics model of Forrester [14] requires well over 500 pieces of primary data and massive computations.

There are two relevant approaches to forecasting that deserve our attention here. The first approach is the classical econometric time-series approach to forecasting [15]. In this approach one looks for statistically meaningful patterns in past data and uses these patterns to predict future patterns. It seems to us that this approach is well-developed and has been applied with some success to certain kinds of software lifecycle modelling [16].

From the standpoint of basic research, however, the traditional forecasting approach is not very satisfying, since it is an admission that the underlying mechanisms have not been understood. Returning to the meteorological analogy, it is an attempt to extend the portents. But that

seems to be the stage of our current understanding of the software lifecycle.

The exact approach to forecasting seems to require many more insights into the various facets of the software lifecycle than we currently have at our disposal. Rather than wait for the software equivalent of Newton (or Galileo, for those who believe that we cannot even measure temperature), we might try to use large-scale computation to build upon the primary data that we can collect. It should be possible to partition a wide variety of programming tasks into discrete, classifiable subtasks that are repeated anew for each project. We can imagine, for example, a catalog of subtasks (such as terminal handlers, hash table routines, and report writers) which are common in various applications software. Notice that we do not claim that these are "off-the-shelf" components -- we merely claim that they must be recreated in approximately the same form for each new job. This catalog will be quite extensive, but it will be conceptually simple to structure and use.

The principle data gathering activity is to determine the cost estimates for each of these subtasks. These costs are influenced by many factors, including the potential application, the skill and experience of the programmers and the restrictions imposed by the programming environment. There are many sources for such estimates. First, there is a great deal of historical data which can be carried forward [19]; after all we do have considerable experience with software projects and this experience can be codified.

Second, we have expert advice concerning the cost of the projects. Third, experimentation can be carried out. Fourth, the cost estimation is from managed projects so feedback can be used to correct prior estimates.

If the data on the primitive tasks that make up the software system is reliable, then the task is to "piece together" a forecast of the total system cost by large-scale computations. By "reliable" we mean that the measurements have an accurate mean and small standard deviation, since in that case the Central Limit Theorem [18] guarantees that the overall estimate will have a small error term (in fact one that grows as the square root of the total number of terms). It is important to distinguish in this approach between using standard cost and estimation data and advocating the use of "standardized software components". Perhaps an analogy to a more familiar cost forecasting problem will make the point more clearly. To estimate the cost of a house, a contractor will consult extensive data sheets on the cost of installing doors (how big?, how many?) and the hundreds of other basic components of a dwelling. These are not prefabricated items, they must be constructed completely from basic specifications and customized for the task at hand, but they are enough alike to permit an accurate assessment of the expected cost of construction. A cost estimate from a builder is pieced together from such estimates.

Now, it is entirely possible that this approach requires inordinate overhead; but there is still room for

applying computational power to chip away at the forecasting problem from historical data. Often, the scale which one uses to assess the software project is even weaker than an ordinal scale: often all that is required is a measure of cohesion between software projects. A manager may only need to know whether the current project is enough like apparently similar projects which have succeeded (or failed) to justify his decision. In this situation the computation needed is a similarity analysis of the important project factors. A large clustering analysis of projected software tasks and historical data may provide such information [10]. The principle task in such an endeavor is to isolate the important factors through data gathering and experimentation.

SUMMARY

We have argued that a major use of software metrics is in the forecasting problem for software projects. By analogy with weather forecasting, we may characterize the current state of knowledge in software forecasting as the gathering of "portents." While these may be useful and sometimes decisive in project management, they are prescientific and qualitative. Further, it seems very unlikely that the portents can be developed into a useful theory of forecasting. To develop scientific forecasting tools a rational way of predicting the future from historical primary data is required. It is also important

that the primary data and the measurements used to obtain it satisfy some basic methodological requirements -- for example, the hypotheses developed from the measurements should be meaningful in the sense implied by measurement theory.

Among the rigorous approaches to the prediction problem we distinguish the statistical and the exact approaches. We specifically reject the notion that such complex phenomena as software lifecycles can be dealt with in a global way using computationally simple "laws". The statistical approach, seeking to predict future events on the basis of historical patterns, seems to be an attractive short range approach to the forecasting problem. There is certainly an extensive body of theory from econometrics and related areas which can be brought to bear on software forecasting. Unfortunately, the statistical approach is a recognition that the underlying mechanisms are not understood. We turn, therefore, to the exact approach. In the exact approach a great deal of effort is spent in attempting to understand -- or to at least quantitatively assess -- the microspic prediction problem. The goal of the exact method is to be able to apply largescale computation to many micro-predictions to synthesize a quantitative forecast. There may even be useful aggregations of statistical and exact techniques which give forecasting models. In both approaches, data gathering is an essential activity; it is therefore important to settle on the fundamental measurements to be performed on the software.

REFERENCES

- [1] Olaf Helmer and Nicholas Rescher, "On the Epistemology of the Inexact Sciences," Rand Corporation Report No. R-353, February, 1950.
- [2] Alan Perlis, Fred Sayward, and Mary Shaw, Unpublished notes on software metrics, April 1980.
- [3] Tim Standish, Notes on Software Metric and ADA, January 1980, Las Vegas, Nevada meeting of ONR Software Metrics Group.
- [4] Jim Browne, "A Philosophy and Justification for Empirical Software Engineering and Software Science," Unpublished Notes 9/13/79.
- [5] R. DeMillo, R. Lipton and A. Perlis, "Social Processes and Proofs of Theorems and Programs, Communications of the ACM, May 1979.
- [6] Philip Thompson, "The Mathematics of Meteorology," in Mathematics Today, edited by Lynn Steen, Springer-Verlag, 1979, pp. 127-152.
- [7] Molly Gleiser, "The First Man to Compute the Weather," Datamation, June 1980, pp. 190-184.
- [8] Alan Perlis, Fred Sayward and Mary Shaw, unpublished Notes.
- [9] Fred Roberts, Measurement Theory, Addison-Wesley, 1979.
- [10] Michael Anderberg, Cluster Analysis for Applications, Academic Press, 1973.
- [11] Norbert Wiener, "A New Theory of Measurement: A Study in the Logic of Mathematics," Proceedings London Math. Society, 1919, pp. 181-205.
- [12] D. Krantz, et al., Foundations of Measurement, vol. 1, Academic Press, 1971.
- [13] Data and Analysis Center for Software, "Quantitative Software Models, MArch 1979.
- [14] J. Forreseter, World Dynamics, 2nd Edition, MIT Press.
- [15] Robert Pindyck, Econometric Models and Economic Forecasts, McGraw-Hill, 1976.
- [16] L. Putnam and A. Fitzsimmons, "Estimating Software Costs," Datamation, September, October and November 1979.

- [18] W. Feller, An Introduction to Probability Theory and its Applications, Volume I, Wiley, 1968.

1. Introduction

Software is more than "source code." Software is used as a generic term for all of the stages gone through in tailoring a program (or programs) to solve some particular problem. The process is non-terminating and the product, software, is evolutionary and shaped both by the nature of its use and the intent of its design. Specification of software is generally incomplete and arrives at a satisfactory state through evolution and use.

Software is subject to a perpetual tension: Being symbolic, it can be perfected, guaranteed, arbitrarily extended, reproduced at almost no cost, completely understood, perfectly managed and eternal. Being symbolic, it can be easily changed, adapted, mishandled, generalized, altered by use and discarded. Far from being reproducible at no cost, the replication of software induces significant extra cost in maintenance and replacement no matter who is responsible for these two activities.

If by software we mean program, every piece of software is (a representation of) an algorithm. Hence the results of the study of algorithms in computer science should suggest methods for controlling and improving software. To a limited degree, computer science has already helped. But the behavior of software is different from that of algorithms. Comparatively we observe:

(1) Software is rarely as precisely specified as algorithms: It is often pointless to speak of software as a map from domain to range and to study its computational complexity.

(2) Unlike algorithms, software changes its intent, to say nothing of its mechanisms, while under specification, design, construction and use.

(3) Software is generally huge, algorithms are described as being precise. Huge objects get that way by processes of accretion and are rarely characterized as being thereby precise.

(4) Software is managed (or mismanaged), algorithms are created, perfected and proven correct.

Issues of performance over a wide range of data may force treatment of some algorithms as software: One speaks of an LP package, not merely the simplex algorithm.

Standardization may allow some software to be treated algorithmically (even as hardware), i.e., the recently announced chip for supporting connection of a processor to the ether net.

It seems reasonable to fix upon a model for software development in which the dynamics of software plays a significant role. The chosen model is the life cycle model in which software is seen as passing through seven stages:

- (1) Requirements Analysis
- (2) Specification
- (3) Design
- (4) Implementation
- (5) Testing, Verification and Integration
- (6) Maintenance and Enhancement
- (7) Replacement and Retirement.

In isolation, these stages occur at progressively later times but feedback from a stage to an earlier one may occur at any time. Revision of specification,

of demand, alteration of requirements arising from use, change in environment and erroneous implementation may interrupt the flow of normal development of spawn sub-processes having their own life cycles.

What are the parameters of flow that will enable us to predict and control the behavior and performance of software as it passes through its life cycle? We must be careful that we not let the symbolic nature of software lead us to expect perfectible methods of prediction and control. Nevertheless, we seek methodologies within whose adherence questions can be asked whose answers will support quantitative comparisons and evaluation of software. Software metrics aid in making choices among options that arise in the life cycle.

2. The Life Cycle

The software life cycle captures one aspect of the evolution of software. The seven stages are attained by all software, but in a complex manner as a set of processes executing overtime. To capture the diversity of process behavior, one is led to think of this passage through the life cycle as the set of execution states of a computation on a computer -- another name for which is the software environment. What plays the role of the instruction code on this computer? We do not yet know this instruction code other than vaguely. However, if we attempt to epitomize software evolution, the set of actions we describe and thereby perform on a piece of software is an example of a program for that computer. The life cycle is the list of the gross procedures we must expect to perform.

The developing interest in software environments is understandable. The proper role of the software environment is much more than a catch-all of

interpreters, source-language editors, debuggers, etc., it is precisely the computer milieu in which software exists as it negotiates its life cycle. One should not underestimate the importance of facile manipulation within a (real) computer of software by functioning software environments at every stage of the former's life cycle. It is in this manipulation that the symbolic nature of software becomes a matter of critical importance, first to control and then (ultimately) to automate the production of software.

3. Issues in Life Cycle Control

In the introduction, a set of typical questions was given associated with each life cycle stage. No questions were assigned to the requirements phase -- the reader can undoubtedly supply a generous list. At each stage the questions

Is it time to go to the next stage?

Is a feedback to an earlier stage (which one?) needed?

must be answered. Sometimes it is appropriate to answer both positively and thereby spawn another software development process.

Let us consider the separate stages in more detail.

Specification

We know what function the software is to perform and have some suggestions on how the functions are to be performed. We are interested in determining (estimates of):

(1) Feasibility. With the resources available (manpower and time) can the software be completed sufficiently before its predicted

obsolescence to warrant its creation.

(2) Generalizability. Can the problem be generalized so that the software becomes more feasible by either or both postponing obsolescence or reducing resource requirement.

(3) Competition. Is the software so critical or the proper design so unclear that competing efforts are worth supporting.

Design

At this stage a detailed formal statement of the problem to be solved and its solution have been prepared, including a development plan for every stage of the life cycle. We are interested in determining:

- (1) What machine configuration to use?
- (2) What language to use?
- (3) Is it possible to incorporate previous work or must everything be built from scratch?
- (4) How will the availability of tool X affect factor Y?
- (5) How close to its limits is the system expected to run?
- (6) What are the potential future enhancements?
- (7) Should the system be all encompassing from which subsystems are carved out as needed, or should the system be a base from which specific systems are built?

Implementation

At this stage there are some questions which need be answered before implementation begins and others which arise during implementation. They include:

- Before: (1) What developmental technology should be used? Should the system be built all at once or should it be constructed through a sequence of executable prototypes?
- (2) What programming discipline should be used? Chief programmer? Cottage industry?
- During and after: (3) Is the project on schedule?
- (4) Is the project on the budget?
- (5) Is the implemented code correct? If not, how close is it to meeting the specification?
- (6) What is the quality of the implemented code? Is it understandable? Will it be maintainable and enhanceable?

Testing

At this stage, the general question to be answered is: "Does the implementation meet the specification?" This usually reduces to questions concerning the implementation's functionality, performance, and useability. The decisions to be made include:

- (1) Should testing be done top down or bottom up?
- (2) Of the available methodologies for testing functionality, performance, and useability, which ones should be used?
- (3) What levels of satisfactory testing are sufficient?
- (4) How will subsequent error reports be handled?
- (5) How well does the testing environment approximate the execution environment?
- (6) How well does the software integrate into the larger system of which it is a part?

Maintenance and Enhancement

Maintenance is similar to testing but different in that the software execution environment has changed from a controlled testing environment to the actual user environment. Enhancement, on the other hand, is a post-release augmentation of the system specifications to meet unsupplied and unforeseen demands. These two very different activities are often linked because they result in a re-release of the system. Note, however, that enhancement causes a feedback cycle of greater distance than does maintenance. With this view, questions concerning maintenance can be thought of as questions concerning testing (see above). Questions concerning enhancement include:

- (1) What is the cost of the enhancement? Is it worthwhile?
- (2) What is the re-release strategy?
- (3) Will the enhancement speed up or delay replacement?
- (4) Will the enhancement disturb or destroy the logical clarity of the system?

Once it has been decided that an enhancement should take place, there is an automatic feedback to the specification life cycle stage.

Replacement and Retirement

Among the questions asked when considering replacement or retirement of a system are:

- (1) Has the problem outgrown the software?
- (2) Has technology moved beyond the software?
- (3) Has a critical support resource for the system become unavailable?

- (4) Would it cost less to redesign and rebuild the system than to maintain and enhance the system?
- (5) How should the system be phased out?
- (6) Do the benefits gained outweigh the costs incurred from changing the language in which the system is written? Changing the machine on which the system runs?
- (7) What are the requirements for the replacement?
- (8) What software is affected by its retirement?

What methodologies grease the passage of software through the life cycle? One might be tempted to argue that it is poor strategy to overdesign and build software with great care for eternal use. Experience shows that some software attains a perfection of design and construction, a complexity of function and a magnificence of size that it assures its own future. The software shifts the burden of evolutionary costs to the milieu. Ultimately the milieu will reject the software -- but the time of retirement can often be delayed considerably. Two methodologies seem attractive: careful use of prototype systems and use of high order languages (HOLs).

4. Prototypes

There are two choices for system development called the "pre-structured" method and the "prototypical" method.

The pre-structured method is the traditional approach in which the final system language, data structures, control structures, and modularization are fixed very early in the life cycle, usually in the design

phase. The pre-structured choices influence enormously the direction of the later life cycle stages and tend to make significant design changes difficult and expensive and, hence, resisted. Nearly all current and past software has been developed using the pre-structured approach.

This method forces a view of software as being "hard," since the costs of change tend to become enormous. Metaphors emphasizing the palpability, the material-like nature of software dominate; that which epitomizes software -- its "soft" character -- is systematically avoided. Management techniques and discipline enforcing methodologies play a key role in the life cycle passage.

In the prototypical approach, on the other hand, the software is seen as being developed through a sequence of executable prototypes with increased functionality, more and more implementation detail and alternative designs being the rationale for the successive prototype systems. Here the design evolves with the implementation and the designer has the advantage of always having an executable model to support design decisions. In a sense the prototypical approach, by "compressing" any single life cycle time span, allows many passes through the life cycle with more and more complex models of the software. Although the potential advantages of using the prototypical approach to software development have only recently begun to be recognized, the value of executable models has long been recognized in other disciplines, e.g., the value of rapid and flexible change has been recognized as crucial in evolutionary success of the MAXIMA system [].

Successor prototypes have an important role to play in identification

of enhancement possibilities and their costs. Their existence provides a source of executable software "history" that aids in maintenance activity and broadens the corps of specialists that must support software "in the field."

These two development approaches differ radically in the language parameter. In the pre-structured method resource efficiency dictates that the final implementation language be chosen early and be the only language used. Most often this language is a computer-oriented language. In the prototypical approach fast and flexible implementation permits, even dictates, that an interpretive language be used for the prototype systems. For it is interpretation, not compilation, which permits incomplete programs and quick response to changes in both control and data structures. Later, for efficiency reasons, some prototype system may require (partial) translation to a language that husbands machine resources better.

The chief question that must be answered is, "In view of the total life cycle cost which approach is more cost effective?" Some advantages and disadvantages of the methods which must be analysed in attempting to find an answer to this question are:

Pre-Structured Advantages

(1) Using one language throughout requires less training of personnel.

(2) Since the design is more or less fixed early in the life cycle, one can concentrate on finding an efficient implementation, proving it correct, arranging personnel schedules, etc.

Prototypical Advantages

- (1) An executable prototype permits the design to evolve, often in ways not envisioned in the initial requirements.
- (2) It is natural to explore competing implementation strategies.
- (3) It is often easy to generalize the requirements and quickly reflect them in a new prototype system.
- (4) Lower average length of life cycle feedback cycle.
- (5) Early prototype systems are potential starting points for evaluating, understanding and performing future maintenance and enhancement.

Pre-structured Disadvantages

- (1) Always working with a static, unproven and unobserved design.
- (2) Changes in the requirements often cause massive redesign of the system.

Prototypical Disadvantages

- (1) Separate languages for the prototypes and the production systems implies a need for either more or for higher qualified personnel.
- (2) The inherent inefficiencies of interpretation must be overcome in going from the final prototype to the production system.
- (3) There is an eighth (possibly costly) life cycle stage of translating from the final prototype to the production system.

Except in extreme cases, for today's software projects, the benefits of the prototypical approach outweigh its disadvantages while the reverse is true of the pre-structured approach. This is supported by the following metaphorical thesis:

The navigation of a software system through the life cycle is a realtime asynchronous process. The system must respond rapidly to the interrupts which will arise due to changing requirements, error corrections, etc. The ease of response materially diminishes the future cost of the software system.

Both development approaches start with flexible models during the initial "at the blackboard" requirements analysis and specification stages. But soon the schism occurs. In the pre-structured approach the informal design is massaged to fit the chosen language -- in the prototypical approach, the informal design is directly reflected and studied in a prototype system. In the pre-structured approach commitment comes early and change is not only painful but often prohibitive -- the prototypical approach is based on rapid easy design changes of almost any kind.

A bias toward the prototypical approach can be expressed in the following informal metaphorical hypotheses:

The prototypical approach forces the inhibition of dramatically fewer life cycle interrupts than the pre-structured approach.

The response time to life cycle interrupts is dramatically faster under the prototypical approach than under the pre-structured approach.

The prototypical approach leads to a more rapid development of software, and the developed software is more reliable, easier to maintain, and easier to enhance.

5. Programming Languages

What factors determine which language should be used for the software project? In general, how do we differentiate between programming languages and how do we connect programming languages to the tasks which are to be

programmed in them?

Many important language and project dependent parameters have been identified. Numerous studies and controlled experiments have been and are being conducted in an effort to determine what, if any, distinguishable effects language choice has on life cycle costs. Nearly all of these studies have centered on comparing languages belonging to what can be called the ALGOL class of languages.

One can be sceptical concerning the conclusions of these works -- mainly because for every result which says, for example, "maintenance is cheaper in language X than in language Y" one can find another result saying "maintenance is cheaper in language Y than in language X." This is not surprising. Indeed, it supports a basic hypothesis:

For any two languages in the same language class (see below) in the long run there is no significant financial gain in using one language over another -- the important parameters are not linguistic but depend on other issues, such as personnel training, software support, familiarity, etc.

There are four classes of language that are in heavy use in today's

software: (1) Machine assembly language.

(2a) ALGOL-like, such as ALGOL 60, FORTRAN, COBOL, and PASCAL.

(2b) ALGOL-like with tasking such as JOVIAL, ALGOL 68, CMS-2, PL/I, and ADA.

(3) Interpretive languages which operate on data structures in parallel such as APL and LISP.

The hypothesis does not mean that, for a given task, any language in class 2b, for instance, is as good as any other language in 2b and one chooses just to suit fancy. On the contrary, there are intra-class language differences. However, the relevant social, educational, management,

and emotional issues dominate the intra-class language differences when it comes to dramatically reducing life cycle costs. Experiments on hypotheses such as

"Maintenance and enhancement is cheaper in ADA than in JOVIAL," and

"Software written in PASCAL is more reliable than software written in FORTRAN"

must lead to inconclusive interpretations.

Clearly this is not the case when comparing languages at the interclass level -- e.g., there is hardly a software manager who would deny that using a class two language rather than assembly language dramatically reduces life cycle costs. This suggests a second hypothesis:

With proper integration into the life cycle, a dramatic decrease in costs can be realized by using a language from a higher level class.

How does one establish such claims? The general lack of theoretical models of the life cycle and programming language characteristics would seem to rule out formal studies.

A proposal has been made by Perlis and Sayward to investigate the use of program mutation as a means of relating programming languages to life cycle costs. A program's mutants are reasonable alternative programs and they have been used as a basis of the mutation analysis testing method. It has been shown experimentally that the level of confidence in the mutation test for FORTRAN programs is proportional to the number of mutants considered in the test. That is, the larger the number of mutants considered, the larger the number of potential errors ruled out.

It is not unreasonable to relate the error proneness of a programming language to the influence on life cycle costs of using the language since error proneness materially affects all later life cycle stages. Consequently, Perlis and Sayward put forward two hypotheses

For a given task, to obtain a given level of testing confidence, there is no significant difference in the number of mutants which must be considered for programming languages in the same language class.

For a given task, to obtain a given level of testing confidence, there is a significant difference in the number of mutants which must be considered for programming languages in different language classes. In particular, there is a dramatic reduction in going from class one to class two and in going from class two to class three.

Progress in programming languages is difficult to characterize, but one trend is clear: Languages show increased ability to produce contextual programs that execute but ignore arbitrary sets of details until such time as they become relevant to a subsequent program 'refinement'. As our collective experience and insight has grown some of these sets have become 'canonized' and their treatment hardened by syntactic and semantic language fixes. We have come to order languages from low to high strictly on the cardinality of their sainthoods.

However it is not cardinality but ease of beautification that matters. This is Teitelman's principle.

If we append to that principle the observation that the invention of good notation or suggestive syntax is found rarely in humans, Occam tells us that, over the long run, that language is highest whose syntax and semantics is simplest subject to Teitelman's principle. Then our machine

assembly languages should surely be highest, but aren't. Why? Backus says it's because our machines are inappropriate -- and he's probably right.

The appropriate machines are those which best fit the languages of class 3, not those of class 2. These machines will not be in large supply for some time. Until then we must invent processors that can map a prototype with the aid of data accumulated on its performance into an efficient program in a lower class language.

Of course the above is hypothesis and experiments must be defined, metrics identified that will support or reject this view of software development.

Notes.

1. The questions arising in the life cycle process were developed jointly with Fred Sayward and Mary Shaw.
2. The hypotheses concerning comparison of languages within and across language classes and the prestructured versus prototype approach to the development of systems are the joint insight of the author and Fred Sayward.
3. References will be appended in the final copy of this paper.

RESOURCE MODELS

Victor R. Basili
University of Maryland

It is important that we have a better understanding of the software development process and be able to control the distribution of resources such as computer time, personnel, and dollars. We are also interested in the effect of various methodologies on the software development process and how they change the distribution of resources. For this reason, we are interested in knowing the ideal resource allocation, how it may be modified to fit the local environment, the effect of various tradeoffs, and what changes should be made in the methodology or environment to minimize resources expenditure.

There has been a fair amount of work towards developing different kinds of resource models. These models vary in what they provide (e.g., total cost, manning schedule) and what factors they use to calculate their estimates. They also vary with regard to the type of formula, parameters, use of previous data, and staffing considerations. In an attempt to characterize the models, we will define the following set of attribute pairs. They can be characterized by the type of formula they use to calculate total effort. A single variable model uses one basic variable as a predictor of effort, while a multi-variable model uses several variables. A model may be static with regard to staffing, which means a constant formula is used to determine staffing levels for each activity, or it may be dynamic, implying staffing level is part of the effort formula itself. Within the multi-variable models, there are various subcategories: adjusted baseline, adjusted table-driven, and multi-parameter equation. The adjusted baseline uses a single variable baseline equation which is adjusted in some way by a set of other

variables. An adjusted table-driven model uses a baseline estimate which is adjusted by a set of variables where the relationships are defined in tables built from historical data. A multi-parameter model contains a base formula which uses several variables. A model may be based upon historical data or derived theoretically. An historical model uses data from previous projects to evaluate the current project and derive the weights and basic formula from analysis of that data. For a theoretical model, the formula is based upon assumptions about such things as how people solve problems. One last categorization is that some models are macro models, which means they are based upon a view of the big picture, while others are micro models in that the effort equation is derived from knowledge of small pieces of information scaled up. We will try to discuss at least one model in each of these categories.

Static single variable models - The most common approach to estimating effort is to make it a function of a single variable, project size (e.g., the number of source instructions or object instructions). The baseline effort equation is of the form

$$\text{EFFORT} = a * \text{SIZE}^b.$$

where a and b are constants. The constants are determined by regression analysis applied to historical data. In an attempt to measure the rate of production of lines of code by project as influenced by a number of product conditions and requirements, Walston and Felix (1) at IBM Federal Systems Division started with this basic model on a data base of 60 projects of 4,000 to 467,000 source lines of code covering an effort of 12 to 11,758 man months. The basic relation they derived was

$$E = 5.2L^{.91}$$

where E is the total effort in man months and L is the size in thousands of lines of delivered source code, including comments. Beside this basic

relationship, other relations were defined. These include the relationships between documentation DOC (in pages) and delivered source lines

$$DOC = 49L^{1.01}$$

project duration D (in calendar months) and lines of code

$$D = 4.1L^{.36}$$

project duration and effort

$$D = 2.47E^{.35}$$

and average staff size S (total staff months of effort/duration) and effort

$$S = .54E^{.6}$$

The constants a and b are not general constants. They are derived from the historical data of the organization (in this case, IBM Federal Systems Division). They are not necessarily transportable to another organization with a different environment. For example, the Software Engineering Laboratory (SEL) on a data base consisting of 15 projects of 1.5 to 112 thousand source lines of code covering efforts of 1.8 to 116 staff months have calculated for their environment the following set of equations (2):

$$E = 1.4L^{.94}$$

$$DOC = 29.5L^{.92}$$

$$D = 4.4L^{.267}$$

$$D = 4.4E^{.26}$$

$$S = 2.3E^{.74}$$

Some other variables, including other ways of counting code, were measured by the Software Engineering Laboratory and the equations derived are given here. Letting DL = number of developed, delivered lines of source code (new code + 20% of reused code), M = number of modules, DM = total number of developed modules (all new or more than 20% new) we have

$$E = 1.58DL^{.96}, \quad E = .063M^{1.186}, \quad E = .19DM^{1.0}, \quad D = 4.6DL^{.28},$$

$$D = 2.0M^{.33}, \quad D = 2.5DM^{.3}, \quad D = 2.0D^{.26}, \quad DOC = 35.7DL^{.92},$$

$$DOC = 1.5M^{1.17}, \quad DOC = 4.8DM^{.99}$$

Most of the SEL equations lie within one standard error of the IBM equation and since the SEL environment involves the development of more standard software (software the organization has experienced in building), the lower effort for more lines of code seems natural. It is also worth noting that the basic effort-lines of code equation is almost linear for the SEL--more linear than the Walston/Felix equation. Remember that the project sizes are in the lower range of the IBM data. Lawrence and Jeffery (3) have studied even smaller projects and discovered that their data fits a straight line quite well, i.e., their baseline effort equation is of the form

$$\text{EFFORT} = a * \text{SIZE} + b$$

where again a and b are constants derived from historical data. The implication here is that the equation becomes more linear as the project sizes decrease.

Static multi-variable models - Another approach to effort estimation is what we will call the static multi-variable model. A resource estimate here is multi-variable because it is based on several parameters, and static because a single effort value is calculated by the model formula. These models fall into several subcategories. Some start with the baseline equation just discussed based on historical data and adjust the initial estimate by a set of variables which attempt to incorporate the effects of important product and process attributes. In other models, the baseline equation itself involves more than one variable.

The models in the adjusted baseline class differ in the set of attributes that they consider important to their application area and development environment, the weights assigned to the attributes, and the constants of the baseline equations.

Walston and Felix (1) calculated a productivity index by choosing 29 variables that showed a significantly high correlation with productivity in their environment. It was suggested that these be used in estimating and were combined in a productivity index

$$I = \sum_1 w_i x_i$$

where I is the productivity index, w_i is a factor weight based upon the productivity change for factor i and $x_i = +1, 0$, or -1 , depending on whether the factor indicates increased, nominal or decreased productivity.

One model that fits into the single-parameter baseline equation with a set of adjusted multipliers is the model of Boehm (4), whose baseline effort estimate relies only upon project size. His set of attributes are grouped under four areas: (1) product--required fault freedom, data base size, product complexity, adaptation from existing software; (2) computer--execution time constraint, machine storage constraint, virtual machine volatility, computer response time; (3) personnel--analyst capability, applications experience, programmer capability, virtual machine experience, programming language experience; (4) project--modern programming practices, use of software tools, required development schedule. For each attribute Boehm gives, a set of ratings ranging from very low to very high and, for most of the attributes, a quantitative measure describing each rating. The ratings are meant to be as objective as possible (hence the quantitative definitions), so that the person who must assign the ratings will have some intuition as to why each attribute could have a significant effect on the total effort.

In two of the cases where quantitative measures are not possible, required fault freedom and product complexity, Boehm provides a chart describing the effect on the development activities or the characteristics of the code corresponding to each rating. Associated with the ratings is a chart of multipliers ranging from about .1 to 1.8. Another model which falls into this category is the model of Doty (5). The Doty model, however, provides a different set of weights for different applications besides two ways to estimate size.

One model which falls into the category of adjusted table-driven is that of Wolverton (6). Here the basic algorithm involves categorizing the software routines. The categories include control, I/O, pre- or post-algorithm processor, algorithm, data management, and time critical routines. Each of these routines has its own cost of development curve, depending upon the degree of difficulty (easy, medium, or hard) and the newness of the application (new or old). The cost is then the number of instructions by category and degree of difficulty times the corresponding cost taken from a table. Another model of this type, but more simplistic, is Aron (7).

The GRC model (8) involves a set of equations derived from historical data and theory for the various activities, several of which are multi-parameter equations of more than one variable. For example, the equation for code development is

$$MM_{CD} = .9773 \times N_{OF}^{1.2583} \times e^{-.08953 \times Y_{EXP}}$$

where MM_{CD} is the baseline staff months for code development task group for a subsystem, N_{OF} = the number of output formats for a subsystem and Y_{exp} is the average years of staff experience in code development. It is worth noting that size of the code is not a factor in this formula. Other formulas exist for the effort involved in analysis and design, system level testing,

documentation installation, training, project control, elapsed time and a reasonable check for the total staff months for the project (MM_{PROJ})

$$MM_{PROJ} = .0218 * ((2 + N_{OF}) * \ln(2 + N_{OF}))^{1.71}$$

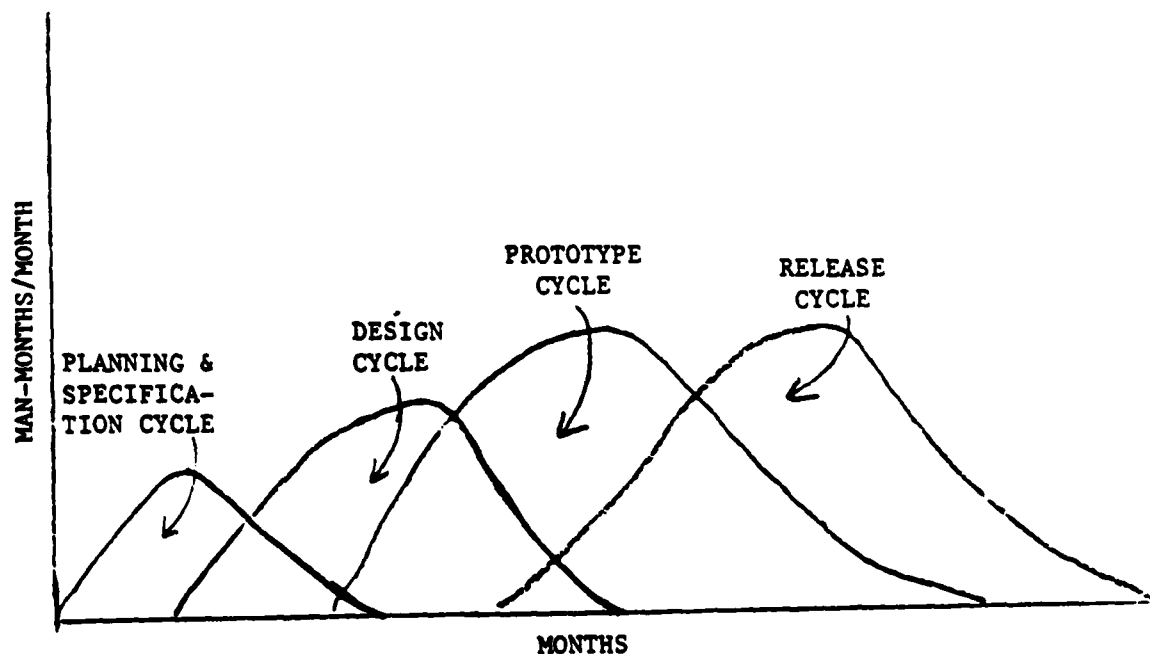
where N_{OF} is as defined above.

Dynamic multi-variable models - Once an effort estimate is made, the next question of concern is how to assign people to the project so that the deadlines for the various development activities will be met. Here again there are basically two approaches: the one empirical, the other theoretical. Each of the methods discussed so far uses the empirical approach which tries to identify the activities which are a part of the development process of a typical project for their software house. Then, using accounting data from past projects, they determine what percentage of the effort was expended on each activity. These percentages serve as a baseline and are intuitively adjusted to meet the expected demands of a new project. For example, in the Wolverton model total cost is allocated into five major subareas: analysis cost (20% of total), design cost (18.7% of total), coding cost (21.7% of total), testing cost (28.3% of total) and documentation cost (11.3% of total). Each of these subarea costs are subdivided again, depending upon the activities in the subareas. In this way, each activity can be staffed according to its individual budget. Allocation of time is determined by history and good management intuition.

The theoretical approach attempts to justify its resource expenditure curve by deriving it from equations which model problem-solving behavior. In other words, the actual resource model lays out the staffing across time and within phases. We will refer to this approach as the dynamic multi-variable model. It is dynamic because the model produces a curve which describes the variation of staffing level across time. The model is multi-factor because it involves more than one parameter.

Two models in this category will be discussed which differ in the assumptions they make. The first model, which is the most widely known and used, is the Putnam model (9).

The model is based on a hardware development model (10) which noted that there are regular patterns of manpower buildup and phase out independent of the type of work done. It is related to the way people solve problems. Thus, each activity could be plotted as a curve which grows and then shrinks with regard to staff effort across time. For example, the cycles in the life of a development engineering project look as follows:



Similar curves were derived by Putnam for software cycles which are: planning, design and implementation, testing and validation, extension, modification and maintenance.

The theoretical basis of the model is that software development is a problem-solving effort and design decision making is the exhaustion process. The various development activities partition the problem space into subspaces corresponding to the various stages (cycles) in the life cycle. A set of assumptions is then made about the problem subset: (1) the number of problems to be solved is finite, (2) the problem-solving effort makes an impact on and defines an environment for the unsolved problem set, (3) a decision removes one unsolved problem from the set (assumes events are random and independent) and (4) the staff size is proportional to the number of problems "ripe" for solution. Because the model is theoretically based (rather than empirically based) some motivation for the equation is given. Consider a set of independent devices under test (unsolved problem set) subject to some environment (the problem-solving effort) which generates shocks (planning and design decisions). The shocks are destructive to the devices under test with some dependent conditional probability distribution $p(t)$ which is random and independent with some rate parameter λ . Assume the distribution is Poisson and let T be a random variable associated with the time interval between shocks

$$\Pr(T > t) = \Pr(\text{no event occurs in interval } (0, t)) \quad (1)$$

where $t = 0$ is the time of the most recent shock

letting $p(t)$ be the conditional probability of a failure given that a shock has occurred and λ be the Poisson rate parameter, then

$$\Pr(T > t) = e^{-\lambda \int_0^t p(x) dx} \quad (2)$$

and

$$\Pr(T \leq t) = 1 - e^{-\lambda \int_0^t p(x) dx} \quad (3)$$

and the p.d.f. associated with (3) is

$$f(t) = \lambda p(t) e^{-\lambda \int_0^t p(x) dx}, \quad t \geq 0$$

This leads to the class of Weibull distributions (known in reliability work) where the physical interpretation that the probability of devices succumbing to destructive shocks is changing with time. Based upon observed data on engineering design projects, a special case of (3) can be used

$$y = f(t) = 1 - e^{-at^2} \quad (4)$$

$$\text{where } p(t) = at \quad (5)$$

$$\text{and } a = \frac{\lambda\alpha}{2} \quad (6)$$

Note that this implies engineers learn to solve problems with an increasing effectiveness (i.e., familiarity with the problems at hand leads to greater insight and sureness). Parameter a consists of an insight generation rate λ and a solution finding factor α . Equation (5) is a special linear case of the family of learning curves: $y = ax^b$.

Equation (4) is then the normalized form of the life cycle equation. By introducing a parameter (K) expressed in terms of effort, we get an effort curve, the integral form of the life cycle equation

$$y = K * (1 - e^{-at^2})$$

where

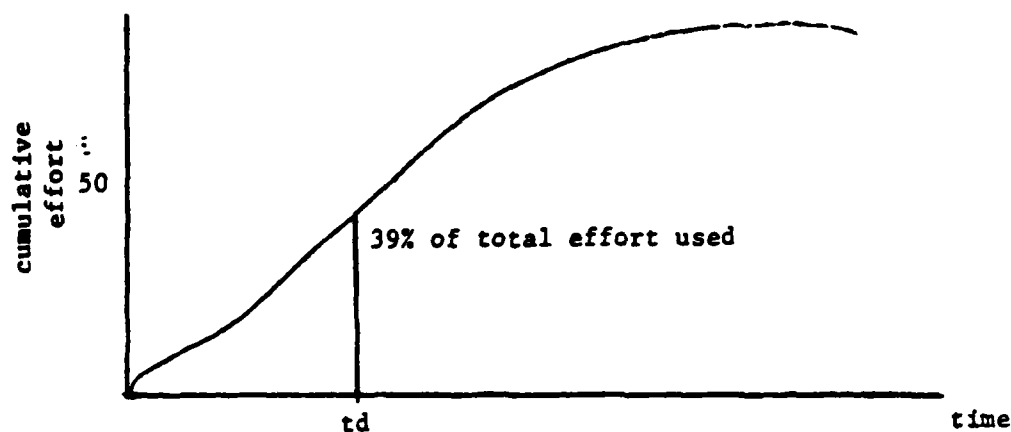
y is the cumulative manpower used through time t

K is the total manpower required by the cycle stated in quantities

related to the time period used as a base, e.g., man-months/month

a is a parameter determined by the time period in which y^1 reaches its maximum value (shape parameter)

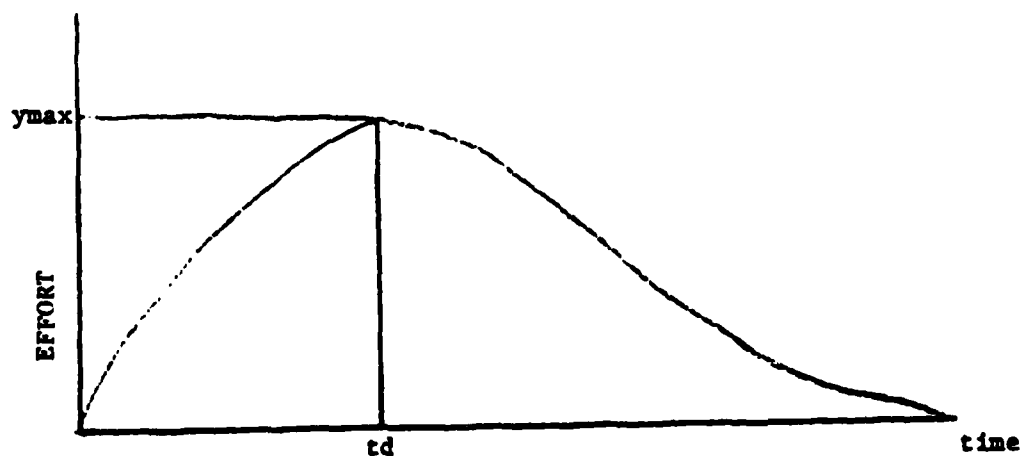
t is time in equal units counted from the start of the cycle



The life cycle equation (derivative form) is

$$y^1 = 2 K a t e^{-a t^2}$$

where y^1 is the manpower required in time period t stated in quantities related to the time period used as a base and K is the total manpower required by the cycle stated in the same units as y^1 .



The curve (called the Rayleigh Curve) represents the manpower buildup. The sum of the individual cycle curves results in a pure Rayleigh shape. Software development is implemented as a functionally homogenous effort (single purpose). The shape parameter a depends upon the point in time at which

y^1 reaches its maximum, i.e.

$$a = 1/2t_d^2$$

where t_d is the time to reach peak effort. Putnam has empirically shown t_d corresponds closely to the design time (time to reach initial operational capability). Substituting for a we can rewrite the life cycle equation as

$$y^1 = \frac{K_2}{t_d} * te^{-t^2/2t_d^2}$$

The equations given are for the entire life cycle. To find development effort only

take

$$y = K * (1 - e^{-at^2})$$

substitute for $a = 1/2t_d^2$

$$y = K * (1 - e^{(-t^2/2t_d^2)})$$

then the development effort is time to t_d

$$y = K * (1 - e^{(-t^2/2t_d^2)})$$

$$= K * (1 - e^{-.5})$$

$$= .3935K$$

or DE = 40% of LC effort

The life cycle and development costs may be calculated by multiplying the cost for that cycle by staff year cost

$$SLC = K * MC$$

where MC = mean cost (in \$) per man year of effort

K = total manpower (in man years) used by project

(Note: the equation neglects computer time, inflation overtime, etc.)

and

$$\text{\$DEV} = \text{MC} * (.3935\text{K}) \approx .4 * \text{\$LC}$$

Putnam found that the ratio $K/(t_d^2)$ has an interesting property. It represents the difficulty of a system in terms of programming effort required to produce it. He defines

$$D = K/(t_d^2)$$

To illustrate how management decisions can influence the difficulty of a project, assume a system size of $K = 400$ MY and $t_d = 3$ years. Then the difficulty $D = 400 / 9 = 44.4$ man years per year squared.

Consider a management decision to cut the life cycle cost of the system by 10%. Now, $K = .9 * (400) = 360$ MY and $D = 360 / 9 = 40$. This results in a 10% decrease in assumed difficulty of the project. This decision assumes the difficulty is less than it really is, and the result is less product.

Now consider the more common case of attempted time compression. Assume management makes a decision to limit the expended effort to 400 MY, but wants the system in 2.5 years instead of 3 years. Now, $K = 400$ MY, $t_d = 2.5$ years, and $D = 400 / 6.25 = 64$ (a 44% increase). The result of shortening the natural development time is a dramatic increase in the system difficulty.

The Putnam model generates some interesting notions. Productivity is related to the difficulty and the state of technology; management cannot arbitrarily increase productivity nor can it reduce development time without increasing difficulty. The tradeoff law shows the cost of trading time for people.

In deriving an alternate model, Parr (11) questions the assumption of the Rayleigh equation that the initially rising work rate is due to the linear

learning curve which governs the skill available for solving problems. He argues that the skill available on a project depends on the resources applied to it and that the assumption confuses the intrinsic constraints on the rate at which software can be developed with management's economically-governed choices about how to respond to these constraints.

As an alternative to this assumption, his model suggests the initial rate of solving problems is governed by how the problems in the project are related, i.e., the dependencies between them. For example, the central phase of development is naturally suited to rapid rates of progress since that is when the largest number of problems are visible. Letting $V(t)$ be the expected size of this set of visible (available for solving) problems at time t , Parr model yields the equation

$$V(t) = \frac{Ae^{-\gamma at}}{(1 + Ae^{-\gamma at})^{\gamma + 1/\gamma}}$$

where

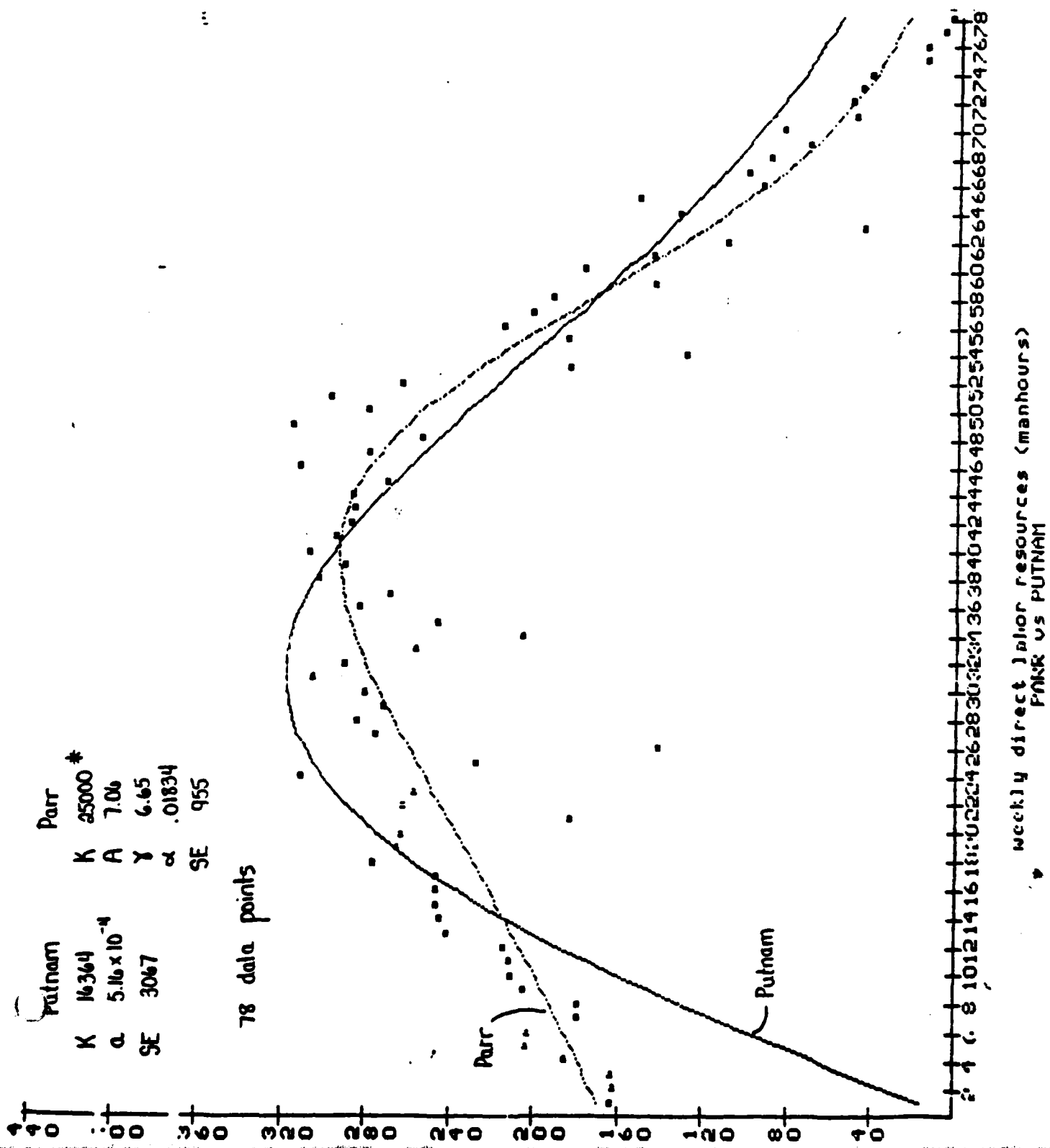
α is the proportionality constant relating the rate of progress and the expected size of the visible set

A is a measure of the amount of work done on the project before the project officially starts

γ is a structuring index which measures how much the development process is formalized and uses modern techniques.

The curve represented by $V(t)$ differs from the Rayleigh/Norden curve for $Y'(t)$ in two important ways. The Rayleigh curve is constrained to go through the origin; the Parr curve is not. Making $Y'(0) = 0$ corresponds to setting an official start date for the project. Before that point, the effort expended on the project is assumed to be minimal. In reality, there is often a good deal of work done before that date, including such activities

as requirements analysis and feasibility studies. In Putnam's environment, these were handled by a separate organization and could be ignored. Another factor that affects the problem space is past experience in the application area, or even more tangible is the influence of design or code taken from past projects. All of these have the effect of structuring the problem space at the beginning, so that more progress can be made early. The Parr curve accounts for this, the Putnam curve does not.



A second distinction between the two curves is the flexibility of where the point of maximum effort can come. By using a structuring index greater than one, this point of maximum effort can be delayed almost to acceptance testing and still be able to drastically reduce effort before project completion. With the Rayleigh curve, a late point of maximum effort constrains the curve to have a slow buildup and almost no decay at the end.

Parr does not say how to estimate the parameters for $V(t)$ in terms of data the project manager would have on hand. This is a problem in doing resource estimation currently, but the model could use the existing resource allocation schedule, based on early data points, to predict the latter part of the curve. The Parr model is only currently being tested on real software for the first time and the results are not yet in. The Rayleigh model, on the other hand, has been used in many environments and has been quite successful on the whole.

Single variable, theoretical - The two previous theoretical models may be thought of as macro models in that the estimate of staffing levels relies on process oriented issues, such as total effort, schedule constraints, and the degree that structured methodology is used. Product oriented issues, such as source code, are not a factor. Most of the other models are less macro oriented in that they consider product characteristics, such as lines of code and input/output formats. In this section, we will discuss another type of theoretical model based upon lower level aspects of the product which we will call a micro model. The particular model discussed here deals with the idea that some basic relationships hold with regard to the number of unique operators and operands we use in solving a problem and the eventual effort and time required for development. This notion was proposed by Halstead as part of his software science (12). Here there is only one basic parameter--size--measured in terms of operators and operands. The model transcends methodology

and environmental factors. Most of the work in this area has dealt with programs or algorithms of module size rather than with entire systems, but that appears to be changing.

In the language of software science, measurable properties of algorithms are

n_1 number of unique or distinct operators in an implementation

n_2 number of unique or distinct operands in an implementation

$f_{1,j}$ number of occurrences of the j^{th} most frequent operator,
 $j = 1, 2, \dots, n_1$

$f_{2,j}$ number of occurrences of the j^{th} most frequent operand,
 $j = 1, 2, \dots, n_2$

then the vocabulary

n of an algorithm is $n = n_1 + n_2$

and the implementation length is

$$N = N_1 + N_2$$

where

$$N_1 = \sum_{j=1}^{n_1} f_{1,j}, \quad N_2 = \sum_{j=1}^{n_2} f_{2,j}, \quad N = \sum_{i=1}^2 \sum_{j=1}^{n_i} f_{ij}$$

Based only on the unique operators and operands, the concept of program length N can be estimated as

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

where

$$\hat{N} \approx N$$

\hat{N} is actually the number of bits necessary to represent all things that exist in the program at least once, i.e., the number of bits necessary to represent a symbol table. Over a large set of programs in different environments, it has been shown that \hat{N} approximates N very well.

To measure the size of an algorithm, software science transcends the variation in language and character set by defining algorithm size (volume) as the minimal number of bits necessary to represent the implementation of the algorithm. For any particular case, there is an absolute minimum length for representing the longest operator or operand name expressed in bits. It depends upon n , e.g., a vocabulary of 8 elements requires 8 different designators, or $\log_2 8$ is the minimal length in bits necessary to represent all individual elements in a program. Thus, a suitable metric for size of any implementation of any algorithm is $V = N \log_2 n$, called volume.

The most succinct form in which an algorithm can be expressed requires a language in which the required operation is already defined and implemented. The potential volume, V^* , is defined as

$$V^* = (N_1^* + N_2^*) \log_2 (n_1^* + n_2^*) \text{ where } V = N \log_2 n$$

but minimal form implies $N_1^* = n_1^*$ and $N_2^* = n_2^*$ because there should be no repetition. The number of operators should consist of one distinct operator for the function name and another to serve as an assignment or grouping symbol $n_1^* = 2$. Thus, $V^* = (2 + n_2^*) \log_2 (2 + n_2^*)$ where n_2^* represents the number of different input/output parameters. Note: V^* is considered a useful measure of an algorithm's content. It is roughly related to the basic GRC model concept of input/output formats.

The level of the implementation of a program is defined as its relation to its most abstract form, V^* , i.e., $L = \frac{V^*}{V}$. $L \leq 1$ and the most succinct expression for an algorithm is a level of 1. $V^* = L \times V$ implies when the volume goes up the level goes down. Since it is hard to calculate V^* , an approximation for L , \hat{L} is calculated directly from an implementation

$$\hat{L} = \frac{2n_2}{n_1 N_2} \sim L. \text{ The reciprocal of level is defined as the difficulty, } D = 1/L,$$

which can be viewed as the amount of redundancy within an implementation.

Based on these primitives, formulas for programming effort (E) and time (T) are derived. Assuming the implementation of an algorithm consists of N selections from a vocabulary of n elements and that the selection is non-random and of the order of a binary search (implying $\log_2 n$ comparisons for the selection of each element), the effort required to generate a program is $N \log_2 n$ mental comparisons (this is equal to the volume (V) of the program). Each mental comparison requires a number of elementary mental discriminations where this number is a measure of the difficulty (D) of the task. Thus, the total number of elementary mental discriminations E required to generate a given program should be $E = V * D = V/L = V^2/V^*$. This says the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language. E has often been used to measure the effort required to comprehend an implementation rather than produce it, i.e., E may be a measure of program clarity.

To calculate the time of development, software science uses the concept of a moment defined by the psychologist Stroud as the time required by the human brain to perform the most elementary discrimination. These moments occurred at a rate of 5 to 20 per second. Denoting moments (or Stroud's number) by S, we have $5 \leq S \leq 20$ per second. Assuming a programmer does not "time share" while solving a problem, and converting the effort equation (which has dimensions of both binary digits and discriminations per unit time) we get $T = \frac{E}{S} = \frac{V}{SL} = \frac{V^2}{SV^*}$. Halstead empirically estimated $S = 18$ for his environment, but this may vary from environment to environment.

Software science metrics have been validated in a variety of environments but predominantly for module size developments.

Other resources - In what has been stated so far, resource expenditure and estimation have been predominantly computed in terms of effort. The formula for cost may be a simple multiplication of the staff months times the average cost of a staff member or it may be more complicated. It may include some difference for the cost of managers versus the cost of programmers versus the cost of support personnel whose role varies across the life cycle (13).

The schedule may be derived based upon historical data, with effort allocated to different activities based upon the known percentages or it may be dictated by the model itself, as with the Rayleigh curve. However, the dynamic models generate what they consider the ideal staffing conditions which may not be the actual ones available. Thus, in fitting actual effort to the estimated or proposed effort, some decisions and tradeoffs must be made.

Computer time is yet another resource. Unfortunately, none of the above models treats this within the same formula. In general, they have a separate formula for computer time again based upon computer use in similar projects. These models vary from a simple table type model (6) to some very sophisticated probability distribution based on reliability modeling for phases of the development, such as testing (14).

Effect of resource model research - In general, modeling a process attempts to explain what is going on in that process by making assumptions about the underlying process and simplifying the environment by removing extraneous and less relevant factors. Modeling presents a viewpoint of the process, in our case the software development process or product by classifying various phenomena, abstracting from reality and isolating the aspects of interest.

Resource modeling can be useful in several ways. It can be used for initial prediction, i.e., given what we know or can guess about a project, it

can be used to predict the effort required to produce the product, the cost, the staffing pattern, the computer time required, etc. The main point is to discover relationships between some set of characteristics that we can estimate or know and those resource elements we wish to find out about. It can be used in predicting the characteristics of the next phase of development from the current phase. We should be able to predict what should happen next, and if it doesn't happen, why not; is it a sign of trouble, etc.?

The most important research use of resource modeling is that it should give us insights into what is going on in the software development process. We can study how different environmental parameters, such as changes in specifications, the methodology or tools used, or the complexity of the requirements can change the pattern of software development. We can use the resource usage patterns to evaluate such things as methodology and tools and learn how to better engineer future developments.

We can use resource data to evaluate our various models of the software development process. For any model, does it explain our behavior and environment? Do the factors (parameters) agree with our environmental parameters and are they calibrated correctly? In this way, we can refine our models of the process and gain deeper insights into the qualitative and quantitative nature of software development.

From the empirical models, we can learn what basic relationships exist between various aspects of the software development process. We can learn what factors affect the development process and what their effects are. From the theoretical models, we can learn if there are better ways to understand the underlying behavior of the process and test out some basic assumptions about group dynamics with regard to software development environments.

References

- (1) Walston, C. and Felix, C., A Method of Programming Measurement and Estimation, IBM Systems Journal 16, Number 1, 1977.
- (2) Freburger, Karl and Basili, Victor, The Software Engineering Laboratory: Relationship Equations, University of Maryland Technical Report TR-764, May 1979.
- (3) Lawrence, M. J. and Jeffery, D. R., Inter-organizational Comparison of Programming Productivity, Department of Information Systems, University of New South Wales, March 1979.
- (4) Boehm, Barry W., Draft of book on Software Engineering Economics, to be published.
- (5) Doty Associates, Inc., Software Cost Estimates Study, Volume 1, RADC TR 77-220, June 1977.
- (6) Wolverton, R., The Cost of Developing Large Scale Software, IEEE Transactions on Computers 23, Number 6, 1974.
- (7) Aron, J., Estimating Resources for Large Programming Systems, NATO Conference on Software Engineering Techniques, Mason Charter, N. Y. 1969.
- (8) Carriere, W. M. and Thibodeau, R., Development of A Logistics Software Cost Estimating Technique for Foreign Military Sales, General Research Corporation, Santa Barbara, California, June 1979.
- (9) Putnam, L., A General Empirical Solution to the Macro Software Sizing and Estimating Problem, IEEE Transactions on Software Engineering 1, Number 2, 1975.
- (10) Norden, Peter V., Useful Tools for Project Management, Management of Production, M. K. Starr (Ed.) Penguin Books, Inc., Baltimore, Maryland, 1970, pp. 77-101
- (11) Parr, Francis N., An Alternative to the Rayleigh Curve Model for Software Development Effort, Transactions on Software Engineering, May 1980.
- (12) Halstead, M., Elements of Software Science, Elsevier North-Holland, New York, 1977.
- (13) Basili, Victor R. and Zelkowitz, Marvin V., Analyzing Medium Scale Software Developments, Third International Conference on Software Engineering, Atlanta, Georgia, May 1978.
- (14) Musa, John D., A Theory of Software Reliability and Its Application, IEEE Transactions on Software Engineering, Vol. SE1, No. 3, pp. 312-327.

INFORMAL PAPER PREPARED FOR
SOFTWARE METRICS PANEL
SPONSORED BY OFFICE OF NAVAL RESEARCH

9-1

June 5, 1980

HIGH LEVEL LANGUAGE METRICS

Jean E. Sammet
IBM Federal Systems Division
Bethesda, Maryland

1. INTRODUCTION

There is a major difference between metrics for one or more high level languages (e.g., FORTRAN, COBOL) and metrics for one or more programs, although these concepts are often -- and erroneously -- used interchangeably. The high level language is clearly a major tool in producing a program, but the metrics are quite different. For example, programs can be measured with respect to their written length, time taken to prepare them, running time, length of object code, complexity, and error rate after completion. Of these measurable factors, two depend primarily on the compiler -- namely running time and length of object code. Two are highly dependent on the ability of the programmer and the nature of the programming problem -- namely time taken to prepare the program and error rate after completion. One is a less objective metric than the others -- namely complexity. Complexity metrics are really beyond the state of

the art right now, although various attempts have been made to define such a measure (e.g. [Bell and Sullivan, 1974; McCabe, 1976]).

Program length seems as if it ought to be an objective measure, but it is not really. The obvious way to measure program length is by the ubiquitous "source lines of code", but that is not clearly defined even within a single language, since free form languages permit numerous statements on a single line. One might think that the number of characters in a program was a completely objective measure, but even here one runs into many questions, e.g., (1) should comments be counted within the program length, (2) should long data names really make a program seem longer than a program which is identical except for using short data names and (3) in a language such as APL which permits overprinting of characters should such overprinted characters be counted as one or two characters? One approach to program metrics is that devised by Halstead [1977].

There will be no further discussion of program metrics, because this paper is meant to deal with language metrics. The measures in this latter area tend to be somewhat more subjective, and harder to define. The main purposes for which language metrics have been used in the past are for language selection and/or comparison, and (separately) for language design. But

there are numerous other issues involved in language metrics besides selection and design. However, they can all be classified under the heading of potential research. Such topics include levels of non-procedurality, deviations from one language to another, functionality of a language, and the relationship of the language to various program measurements. This paper discusses each of these very briefly and informally, and mentions the difficulties in carrying out experiments in language areas.

The term "language" or "programming language" as used in this paper refers only to high level languages such as FORTRAN or COBOL. There is no discussion in this paper of assembly languages nor of such "midway" languages as PL/360.

2. NUMERICAL APPROACHES TO LANGUAGE SELECTION/COMPARISON

Language selection involves technical and non-technical issues. An example of the former is the ability of the language to handle subscripts (e.g., how many, what type of subscript expressions) or to permit elaborate data layouts. An example of the latter is the amount of training needed by the people who are going to use the language. There are some items whose classification is unclear -- the most notable one is the compiler. Most people ignore this important distinction between the technical and the non-technical issues, and get into intellectual and

practical difficulty as a result. However, in all cases, the application area for which the language is to be used should be the major consideration for technical evaluation.

There are two basic numerical methods for doing language selection: numerical scoring against requirements, and benchmarking.

2.1 Numerical Scoring of Languages

Various attempts have been made to make the process of language selection more abstract, or at least more visible and objective and less dependent on handwaving and arbitrary human judgment. If the requirements of the problem for which the language is to be chosen are specified at the beginning, then a simple scoring technique can be used, although it is often difficult to apply. The technique simply assigns each requirement a weighting factor to indicate its relative importance among all the other requirements. Then each language is evaluated for its ability to deal with that particular requirement. The numbers in both cases are usually normalized, and then they are cross multiplied and a final score is obtained. For a more detailed explanation of this method, see [Browne et al, 1970; Sammet, 1971]. The technique has been used in real situations (e.g., [Browne et al, 1970]).

Now nobody, least of all this author, would claim that such a method eliminates all arbitrariness; people can use unreasonable assignments of weights on both sides (i.e., both the importance of the requirements and the evaluation of the language against the requirements) to enable them to reach a predetermined conclusion. However, this method has the advantage of at least permitting all of these biases to be clearly visible to other people who will review the scoring process.

In some cases, a prior step must be taken before the above technique can be used; this occurs if the requirements themselves are not clearly determined at the outset. The requirements themselves could be determined by a similar type of weighted scoring method, or by some nonnumeric method of group decision. Again, there is no guarantee of objectivity, but at least the biases will become visible if a scoring technique is used here. The most extensive effort in comparing languages against requirements was the DOD sponsored work in evaluating about 20 languages against the TINMAN requirements [Amoroso et al, 1977].

2.2 Benchmarking

The basic principal of benchmarking is to choose problems and then write programs in the languages which are under consideration. The major difficulty of course is to select a fair set of

problems which is representative of the application to be implemented. There are then three types of benchmarking that can be done. The first involves merely writing the programs on paper and then measuring both the length of the program and the time it takes a person to write it. Not much more can be done with regard to measuring paper programs of that kind.

A second possibility is to actually execute the programs and then measure those characteristics which are relevant to the eventual application. For example, one can measure the length of object code, the execution time for the running program, and any other environmental factors that are relevant. If a compiler which is to be used for the final project is actually available for these benchmarks, and if the problems are validly selected, then significant results can be obtained this way.

A third way of measuring for benchmarking is to measure the characteristics of the programs using measures such as those defined by Halstead [1977] or McCabe [1976]. The conclusions to be drawn from such measures will depend on how much one believes in either or both of those measurements. Such a discussion is beyond the scope of this paper.

The earliest of the benchmarking studies seems to have occurred in the early 1960s although the reference is currently unavailable. Certainly one of the early thorough attempts at comparing languages was [Rubey, 1968] which used benchmark problems to compare PL/I to COBOL, FORTRAN, JOVIAL.

There are several difficulties with considering benchmarking as a very good metric. First, the capabilities of individual programmers tend to be more significant than the qualities of the languages themselves; there are generally ways around this by having the same programmers code problems in several languages, but then there is a difficulty in the person having more familiarity with the problem when it is coded in the second language. A major -- but often very subtle difficulty -- arises in the actual choice of problems to be used for the benchmarks. It is entirely possible to choose problems which favor one language over another, and this is sometimes done unconsciously. Furthermore, this bias (whether deliberate or accidental) is often impossible to discover unless some objective person is expert in all the problems and all the languages under consideration.

()

3. METRICS FOR LANGUAGE DESIGN

In order to apply metrics to language design, there are two basic numerical approaches. The first involves feature comparison, and the second involves program metrics.

In order to do a feature comparison of the desirability of various language designs, one can carry out human experiments to compare features with respect to some desired criteria (e.g., readability, brevity). A number of people have conducted experiments in this area, e.g., see [Gannon and Horning, 1975; Gannon, 1977]. These and many other experiments (including some of his own) are described in [Schneiderman, 1980]. Among the most interesting of these is Gannon [1977]; he designed two different simple languages and had two sets of programmers write programs. The language design issue being investigated was typed versus typeless languages. The measurements were made on the number and types of errors in the programs.

Another way of dealing with feature comparisons is by various linguistic approaches, e.g., examination of concrete and abstract syntax, types of grammar with various features. In particular, the size of the grammar can provide a metric for a language.

A third factor is the effect of various features on compilers. In this context, one determines which features or combinations thereof are easier (or more difficult) to implement with respect to a particular compiler design.

As a fourth technique it is always possible to compare language features against already existing languages according to some specific criteria (e.g., number of subscripts allowed, presence of recursion) and reach some determination. This approach could provide relative metrics between two languages rather than an absolute measure for a language.

One can also use program metrics (e.g., Halstead) to measure one or more language designs. Thus if one is a believer in some of these program metrics, one could try writing programs in several alternate language designs and see what conclusions could be reached.

4. POTENTIAL AREAS OF FUTURE RESEARCH

There are a number of fruitful areas for research in the area of language metrics.

4.1 Technical vs Non-Technical Issues

Strengthen the understanding and the measurement of the differences between technical and non-technical issues. This would then enable the language selection to be done more abstractly. At best the results in this area will be subjective and relative, e.g., is the compiler a technical or a non-technical issue with respect to the language? In my opinion it is non-technical, but others will surely disagree, which simply reemphasizes the subjective nature of the issue.

4.2 Measurement of Language Deviation

Develop methods of measuring the deviation of one language from another. This has implications which range all the way from the purely scientific to the practical area of contracting. Thus, if one knows the level or amount of deviation of one language from another, it may be possible to draw certain conclusions about programs to be written in one versus in the other. From a contractual point of view, it may become important to know whether or not a language is a major or minor "deviation" from some base language. Everybody uses the terms "dialect" and "language-L-like", but nobody has ever supplied a scientific meaning to these terms, and metrics should be developed to give meaning to the terms. A primitive attempt at starting this work

is given in [Sammet, 1971]. This seems to be a very fruitful area of research and a few good Ph.D theses might solve this problem.

4.3 Measure of Non-Proceduralality

Develop measures of "non-proceduralality". It is agreed by virtually every knowledgeable person in the field that non-proceduralness is a relative term which changes as the state of the art changes. (See a discussion in [Leavenworth and Sammet, 1974]). Since one objective of research in languages is to allow the user to specify as few of the details about his problem as possible, metrics of non-procedurality will enable us to measure our own progress towards this goal. In addition, people will have some choice about how many details about a particular problem they need to specify, by choosing a language at the right level of non-procedurality. Thus, the more non-procedural language will permit the user to specify fewer details. This is a very difficult -- although very important -- research topic. It is strongly related to the problem of measuring functionality discussed in the next section.

4.4 Measures of Programmer Productivity Via Functionality

Develop a good measure for programmer productivity. Clearly this is one of the most crucial issues for which metrics are

needed, but also seems to be the hardest. The only thing that is clear is that source lines of code is a poor measure, and yet it seems to be the most prevalent one. What is desperately needed is a measure of functionality for a language and for a program written in the language. From that we could derive programmer productivity. As an example, it is clear that a call to a subroutine provides more function than a simple GO TO. On the other hand, does a single loop statement provide more function than a single CALL? And in which case has the programmer accomplished more?

Research is needed for both the language and the programs written in that language. With the technology of 1980, the use of a "statement count" is a more accurate measure of function accomplished in a program than a count of "physical lines of code" in the program. However, even the definition of a statement for these purposes needs research to have consistency from one language to another. For example, how many statements (for purposes of functionality and productivity) are there in IF A THEN B ELSE C? And does that answer change if A or B or C are themselves "IF statements" or does the answer depend on the language?

4.5 Measuring Languages vs Measuring Programs

Develop techniques to numerically bridge the gap between the measures of a language and measures of programs written in that language. Right now, these measurements are frequently mixed up or interchanged or not even recognized as being separate concepts. A key issue is the "source lines of code" discussed in the previous section, which may have different measures depending on whether we are talking about the language or a program written in the language. This is as much an educational problem as a research problem.

4.6 Measuring Definitional Techniques

Develop measures of definitional techniques. It is clear that the syntactical definitional technique which is used actually has a significant effect on the language if the syntactic method is chosen at the beginning. But we have no way of measuring this. This is a very difficult topic, but fortunately an area of relatively little importance.

4.7 Languages and the Life Cycle

Develop measures of a language, and its usefulness, over the whole life cycle of a program. This probably involves consideration of some languages besides the classical high level languages, although measures of the latter in this context are

also of interest. Some languages may be useful for expressing specifications or designs but poor for doing programming. Other languages might be good for debugging and maintenance but poor for writing programs quickly. Thus we would like to be able to measure the appropriateness of a language with respect to each portion of the life cycle; clearly such a metric would be a major factor in language selection (as discussed in an earlier section).

4.8 Languages and Applications

Develop techniques for measuring the applicability of a language to a particular application or class thereof. As discussed in an earlier section, this is generally done by evaluating specific technical and non-technical features of a language in a subjective way. But perhaps there is some way of making this measurement more objective.

5. POTENTIAL EXPERIMENTATION

Unlike some other areas, it is difficult to suggest useful valid experiments in the area of language metrics. As with many other aspects of measuring software, the individual differences of programmers, or the individual differences in the applications involved may swamp the element(s) being measured.

Specifically with regard to languages, one of the obvious types of experiments involves language features (e.g., [Gannon, 1977]). However, the issue here may be of the background language within which the feature is being tested rather than the feature itself. Thus, an issue of strong typing in a language, versus weak typing or typeless languages cannot be separated entirely from the surrounding language.

Another fruitful area for experimentation is the cost occurring from the presence or absence of a particular feature in a language. Such costs occur in learning the language, the compiler, the maintenance of the programs in the language, and potential needs for portability. It is important to note that this experimentation applies equally well to presence and absence of a feature. As a simple example, the ability to allow any arithmetic expression as a subscript immediately leads to the need for rules about a floating point value as a subscript. Is the (apparent) generality of the rule just stated a help or a hindrance in learning the language and writing accurate maintainable programs? Perhaps some experiments about major features could be run, but this would probably be of intellectual interest only. Usually, time and economics don't permit this type of work to be done while the language is being designed!

REFERENCES

- Amoroso, S., et al. Language Evaluation Coordinating Committee Report to the High Order Language Working Group (HOLWG). AD-A037634, Jan. 1977, 2617 pages.
- Bell, D. E. and Sullivan, J. E. Further Investigations into the Complexity of Software. MITRE Technical Report MTR-2874, Vol. II, Bedford, Massachusetts, June 1974.
- Browne, P. H., et al. DATA PROCESSING TECHNOLOGIES, VOLUME I - HIGH-LEVEL LANGUAGE EVALUATION. Teledyne Brown Engineering, Huntsville, Alabama, Army Contract No. DAHC60-69-C-0037, May 1970.
- Gannon, J. D. "An Experimental Evaluation of Data Type Conventions", CACM, Vol. 20, No. 8, August 1977, pp. 584-595.
- Gannon, J. D. and Horning, J. J. "Language Design for Programming Reliability", IEEE Trans. Software Eng., Vol. 1, No. 2, June 1975, pp. 179 - 191.
- Halstead, M. Elements of Software Science, Elsevier, New York, New York 1977.
- Leavenworth, B. and Sammet, J. E. "An Overview of NonProcedural Languages" Proc. ACM SIGPLAN Sympos. on Very High Level Languages SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 1 - 12.
- McCabe, T. J. "A Complexity Measure", IEEE Trans. Software Eng., Vol. 2, No. 6, Dec. 1976, pp. 308 - 320.
- Rubey, R. J., et al. Comparative Evaluation of PL/I. Logicon Inc., San Pedro, California, AD-669096, April 1968.
- Sammet, J. E. "Problems in, and a Pragmatic Approach to Programming Language Measurement", AFIPS Fall Joint Computer Conf., 1971, pp. 243 - 251.
- Schneiderman, B. SOFTWARE PSYCHOLOGY: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., Cambridge, Massachusetts 1980.

PERFORMANCE EVALUATION: A SOFTWARE METRICS SUCCESS STORY

by W. C. Lynch and J. C. Browne

June 2, 1980

ABSTRACT

Performance evaluation has generally accepted metrics for both externally observable aspects of performance (response times, rates of completion for externally defined units of work, etc.) and for the performance of intermediate abstract machines (resource consumption for execution of a given externally defined unit of work, etc.). Values for these metrics are, however, often strongly context dependent. Performance evaluation has generated a scientific approach to the development of the field. There is an effort to extract and define invariant principles and there is a systematic hierarchic structuring of performance models to relate concepts and metrics at different levels of abstraction. The fundamental problem is the absence of operational (software engineering) procedures which will yield software systems with desired values for given metrics. The principal research directions recommended for increased emphasis are those in support of this problem are the development of invariant principles which establish direct relations between elementary units of work (abstract machine work) and externally defined units of work, and the development of technology for increasing the effectiveness and efficiency of the performance evaluation process. There is a strong need for the development of abstract machine models appropriate for different types of workload elements. An associated unmet requirement is a capability for describing and specifying performance characteristics for these abstract machines.

UNIVERSITY OF TEXAS AT AUSTIN
Painter Hall 3.28 / Austin, Texas 78712

XEROX

SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill RD. / Palo Alto / California 94304

1.0 DEFINITION AND OVERVIEW

Performance evaluation is determining and evaluating in context the completion time and resource consumption of processes executing specified tasks on abstract machines. The early emphasis of performance evaluation (when hardware was the expensive element of a computer system) was on the definition and measurement of (incomplete) sets of internal (efficiency oriented) metrics for resource utilization. The goal was to maximize the efficiency of the computer system itself. Current emphasis is on the definition, measurement, and prediction of external metrics (such as response time and work throughputs). The goal is to maximize effectiveness and productivity of the total system.

An abstract machine may be a hardware realized machine or an abstract machine defined in terms of abstract operations such as the comparisons of a sort algorithm or the multiplies of matrix algorithms. A process may be a complex software system such as a data base transaction system operating under an operating system on a large mainframe or it may be a simple sort algorithm on a dedicated microcomputer. The task may be large, e.g., determine the flux of neutrons 1,000 kilometers from the epicenter of a given thermonuclear detonation, 30 degrees from the direction of a 10 mph prevailing wind, or small, such as sorting a short and almost sequential list of names. We confine ourselves here to considering performance evaluation of substantial software systems on non-trivial (although perhaps abstract) computer systems. The performance evaluation of systems executing elementary units of work appears here only as components of a hierarchic model, used systematically to lead to the evaluation of the performance of a substantial software system. The study of processing of elementary units of work, algorithm analysis, is a substantial sub-branch of computer science in its own right. It is logically a part of performance evaluation but is in practice largely executed by complexity theorists. The interested reader is referred to the classical books of Knuth [KNU68,69,73], Horowitz and Sahni [HOR78] or Aho, Hopcroft and Ullman [AHO74] for substantive works on algorithm analysis.

The very rapid rise in the role of person/machine interactions in work environments makes the external metrics of responsiveness one of the most important criteria by which a software system can be evaluated. Lack of human scale responsiveness can have significant impact on the productivity of persons using a computer system and may lead to the propagation of increased cost throughout an organization. Factors such as long delivery times and costs of housing, operations and communications as well as hardware costs make the prediction of performance of software systems in advance of implementation a significant problem. The dramatic rise in the integration of computer systems into communication systems, monitoring systems and control systems (the generic class of imbedded computer systems) is also increasing the significance of performance metrics, both internal and external. The space and power limitations of process control environments often limit hardware capacity enhancement and force consideration again of efficiency in the use of resources for imbedded systems. There have in the past few years appeared several texts and monographs on performance evaluation. These include [DRU73], [HEL75] and [FER78]. These books offer introduction to techniques and to representative applications.

2.0 PERFORMANCE METRICS

The metrics for the evaluation of performance of large software systems are disjoint in concept from the metrics for the evaluation of other properties such as maintainability, understandability, etc. The "ability" properties are difficult to define. Their definition cannot now be based on independent concepts. The rates at which a computer system delivers completed externally defined units of work are definable and measurable. External performance metrics (e.g., transaction or interactive response time, work thruput) are thus simple and well defined and understood. Complexity arises from the fact that a performance metric value is not an intrinsic property of the software sub-system alone. Many metrics are meaningful only in the context of a specific hardware configuration and a specific workload to be executed by the software system. Thus

interpretation of given metric values is subjective and context dependent. Note that performance evaluation has developed a hierarchically structured model where external metrics from one level become parameters for higher levels (see Section 3, following). The internal metrics of resource consumption by a given external unit of work or device utilizations under a given load of external work are also in principle directly measurable and have readily interpretable value sets. There is still here subjectivity and context dependence due to lack of comparability between physically realized abstract machines.

There are two other aspects of performance evaluation which need comment. One is the concept of cost effectiveness to obtain a given level for an important metric such as responsiveness. It is here that an additional connection is made between internal and external metrics. It is often possible to quantify costs in terms of resource power or resource availability. Another is the concept of total system effectiveness where system definition includes the users who interface directly with the system. This area of the quality of human interfaces is an extremely significant problem in its own right. This article will confine itself only to the time responsiveness aspects of the human interface.

3.0 THE STATUS OF PERFORMANCE EVALUATION

The paper by Browne and Shaw [BRO80] in this volume develops a paradigm for science in the context of software systems. This paradigm points to the need for invariant principles as a basis for a science and to the almost universal presence of a hierarchic concept structure relating more complex objects to less complex objects.

Performance evaluation has two serious streams of research aimed at development of basic invariant principles. The "software physics" work of Kolence [KOL72], Helleman [HEL72], etc. is based upon information theory concepts. The theme of this work is to establish operational concepts for information transformations similar to the entropy concept in physics or information theory. Unit operations on unit objects would in these models create precisely known changes in state variables. The operational analysis work of Buzen [BUZ76] and Buzen and Denning [BUZ78] is a second and fundamental approach. Operational analysis is a sort of cosmology for software whereby the relationships between the macro metrics of a system are developed without resorting to the process through integration through micro metrics and models.

Performance evaluation also has a set of structural concepts and relationships which enable the hierarchic integration of individual unit models into a parametric system model. There are two sub-streams -- the process/event model [FRA75, MAC75] which was developed for simulation languages and the network of queues model [KL175, GRA78] which was picked up by computer science as an analytical tool in the 1960's and has been extensively developed in the computer system problem area.

There is a third stream of work focusing on the development of a whole catalog of unit models which act as atoms in the hierarchic models developed by simulation or queuing network analysis. Included in this stream is the development of a large collection of single queue models whose integration into a system is addressed by the network queuing models referred to above. Some of the non-queuing models are quite simple but some (eg., models of locality of reference effects) [SPI77] are not. Algorithm analysis is currently the dominant source of unit models.

The interdependence of software system performance evaluation with hardware and workload concepts is easily seen in the queueing models. The queue/server pairs generally represent hardware devices (and their software schedulers) while the resource consuming units are labelled jobs. The actual service patterns (including the value of the response time metric) are controlled both by the system's internal service facilities (hardware and software) and by the external requests for service presented by the workload. Figure 1 is a simple queueing network model of a

computer system including a processing unit (CPU), I/O devices (DISK) and user terminals (TTY). The units of work can be modelled as originating in the TTY server. There may be several types of requests, each with a distinct requirement for CPU and DISK service. The arcs in the figure represent the flow of job requests from device to device. The P_{ij} labeling the arcs are the probability that a job of type i takes branch j .

The dependence of software system performance evaluation on abstract machine execution and workload pervades through all hierarchic levels for system models. Algorithm analysis is performance evaluation of a simple system on an abstract machine under a given workload. A sort algorithm is, for example, said to require $O(n \log n)$ comparisons if the elements being collated are initially in a random sequence. The abstract execution machine is a comparison machine while the significant property of the workload, the initial state of the element sequence, is specified. This structure carries to the much higher levels of abstraction of queueing network models. A queue may represent a disk or a CPU while the service patterns of the servers are determined jointly by the specification of the workload, the application algorithms and the scheduling disciplines for the software system.

Courtois [COU77] discusses decomposability and hierarchic structuring in general and mathematical terms. Browne, et al [BRO75] give a case study of the application of hierarchic modeling based upon queueing networks. Chandy, Herzog and Woo [CHA75] develop the formal basis for analytic determination of the single queue equivalent to a network of queues.

In a multi-level hierarchic system, the metrics of the lower level abstract machine are available as and become the parameters of the higher layers of the system. This systematic layer-by-layer construction methodology focuses attention on those metrics of the lower level abstract machine which are the required parameters for the performance evaluation of the upper levels.

This provides another explanation for the focus on response times as important metrics for computer systems. The response times of the lower layers become service times for the upper layers. In a person/machine system, the response times of the machine subsystem determine a subset of the service times in the upper, human interfacing system.

Performance evaluation can thus, in several senses, be regarded as conforming to the paradigm for a software science. There are, albeit not widely accepted, invariant principles which can be related to observables and used to define fundamental properties of scale for performance. There is a theoretical structure (models of systems) which establishes useful input/output relationships which can be used both for an ex-post-facto explanation of the observed behavior of an existing system and also for the prediction of future behavior of systems perturbed from base line models.

While there is now emerging a methodology for the prediction of performance of a software system in a given hardware and workload context we do not yet have good synthesis or design capabilities. We are not at present able to achieve the software engineering of performance properties. There do not at present exist well established methodologies for the development of software systems which, put in the context of a hardware configuration and a workload, will deliver specific performance metric values. However, work in this area of design and synthesis is now in progress. Attention is called to [FER78] and to the papers of Smith and Browne [SM179a,79b,80], of Sanguinetti [SAN79] and of Shaw [SHA79] for several different aspects of this problem.

It is not difficult to identify the impediments to the achievement of a design methodology. A design methodology must have as its first step an understanding of the design space (or a useful subset of the design space) as a function of the system specifications. The second step of a methodology must be a set of procedures which select a specific design from the space. Each potential design selected can then be evaluated by the analysis procedures described preceding. Additional procedures can then be used to sequentially select better designs until an acceptable one is achieved.

At present, we have only a modest amount of research attempting to describe the design space for synthesis of large software systems with most of the existing work concentrated in the data base area. The procedures which systems programmers use to select initial and improved designs is not well understood and not nearly ripe for automation.

The relationship of performance evaluation analysis to computer system design seems quite analogous to the relationship of finite element analysis to the design of structures in civil engineering. In the latter case, an adequate understanding of the design space and of procedures for selecting improved designs has permitted the substantial automation of all kinds of structures from bridges to airframes. It has also improved our knowledge of the relationships between requirements and the value of metrics in those areas.

4.0 RESEARCH AND DEVELOPMENT IN THE PERFORMANCE EVALUATION AREA

A fundamental problem in performance evaluation is the requirement for a methodology which can predict both internal and external performance metrics from specifications rather than from base line cases. In imbedded systems, in particular, it is important to know performance metrics before a commitment is made to physically engineer a particular set of resources into a constrained environment. This general problem leads to a number of problems which must be solved to support this constructive capability.

First, our analytic capability still has some shortcomings in both unit models and the analysis of hierarchic systems integration. More realistic unit models need to be constructed, particularly addressing the question of service costs due to information accessing and due to algorithm setup times. Better unit models describing memory space requirements are also needed and most of the models could be improved by including parametric dependencies on workload characteristics. With respect to hierarchic systems integration, the representational capabilities of rapidly soluble system models, such as queueing network models, need to be greatly extended. Additional capabilities which are required include capabilities for rapid solution of finite queues and extended sets of scheduling disciplines. Attention also needs to be paid to the development of model structures appropriate to asynchronous (data flow control) concurrent executions. (See the Petri net [PET77] based models of Noe and co-workers [NOE73], [CRO75]; also [BRY79].)

Second, we need to develop design (or design sketch) methodologies which are capable of integrating the existing analysis capabilities if we wish to establish causal relationships between specific system requirements and specific values of system metrics. As noted before, this will require a substantial increase in our fundamental knowledge of the topology of the design space as a function of the system specification. Improvement in this area depends upon development of better unit model structures. It will also require a substantial increase in our fundamental knowledge of procedures for selecting good (or improved) designs from the design space. This is a very general problem pervading all of software development.

Third, as we develop fundamental invariant principles at various levels of the system, we will be able to validate model system representations against the invariant principles since they establish direct relationships between requirements and performance metrics. The current state of affairs is, however, that both the information theory and operational analysis approaches are currently in a state of flux and are not accepted widely as fundamental at their respective levels of abstraction. Continuing basic research in this area is sorely needed as the payoffs for any progress in this area are quite large.

References

- [AHO74] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS. (Addison-Wesley, Reading, 1974).
- [BRO75] Browne, J. C., Chandy, K. M., Brown, R. M., Keller, T. W., Towsley, D. F. and Dissly, C. W., "Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems", Proc. IEEE 63, 966-977 (1975).
- [BRO80] Browne, J. C. and Shaw, M., "Science for Software Evaluation", (this volume).
- [BRY79] Bryant, R. F. and Dennis, J. B., "Concurrent Programming in RESEARCH DIRECTIONS IN SOFTWARE TECHNOLOGY", (MIT Press, Cambridge, 1979).
- [BUZ76] Buzen, J. P., "Fundamental Laws of Computer Systems Performance", Proc. of Intl. Symp. on Computer Perf. Modeling, Meas. and Eval., Cambridge, Mass., March 1976, 200-210.
- [BUZ78] Buzen, J. P. and Denning, P. J., "The Operational Analysis of Queueing Network Models", Computing Surveys, 10, 225-262 (1978).
- [CHA75] Chandy, K. M., Herzog, U. and Woo, L., "Parametric Analysis of Queueing Networks", IBM, J. Res. Develop., 19, 36-42 (1975).
- [COU77] Coutois, D. J., DECOMPOSABILITY: QUEUEING AND COMPUTER SYSTEM APPLICATIONS. (Academic Press, NY, 1977).
- [CRO75] Crowley, C. P. and Noe, J. D., "Interactive Graphics Simulation Using Modified Petri Nets", Proc. on Simulation of Computer Systems, Boulder, Colorado, August 1975, 177-185.
- [DRU73] Drummond, M. E., EVALUATION AND MEASUREMENT TECHNIQUES FOR DIGITAL COMPUTERS. (Prentice-Hall, Englewood Cliffs, 1973).
- [FER78] Ferrari, D., COMPUTER SYSTEM PERFORMANCE EVALUATION. (Prentice-Hall, Englewood Cliffs, 1978).
- [FRA75] Franta, W. R., PROCESS/EVENT DRIVEN SIMULATION. American-Elsevier, 1975).
- [GRA78] Graham, G. S., "Guest Editors Overview: Queueing Network Models of Computer Systems Performance", Computing Surveys, 10, 219-224 (1978). This issue of Computing Surveys is dedicated to queueing network models of computer systems.
- [HEL72] Hellerman, L., "A Measure of Computational Work", IEEE Trans. Comp., 21, 439-446 (1972).
- [HEL75] Hellerman, H. and Conroy, T. F., COMPUTER SYSTEM PERFORMANCE. (McGraw-Hill, New York, 1975).
- [HOR78] Horowitz, E. and Sahni, S., FUNDAMENTALS OF COMPUTER ALGORITHMS. (Computer Science Press, Inc., Potomac, MD, 1978).
- [KLI75] Kleinrock, L., QUEUEING SYSTEMS: VOL. 1. THEORY and VOL. 2. COMPUTER APPLICATIONS. (John Wiley and Sons, 1975).

- [KNU68] Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, VOL. 1: FUNDAMENTAL ALGORITHMS. (Addison-Wesley, Reading, 1968).
- [KNU69] Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, VOL. 2: SEMI-NUMERICAL ALGORITHMS. (Addison-Wesley, Reading, 1969).
- [KNU73] Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, VOL. 3: SORTING AND SEARCHING. (Addison-Wesley, Reading, 1973).
- [KOL72] Kolence, K., "Software Physics and Computer Performance Measurements", Proc. ACM Natl. Conf. 25, 1024-1040 (1972).
- [NOE73] Noe, J. D. and Nutt, G. J., "Macro E-nets for Representation of Parallel Systems", IEEE Trans. Comp. C-22, 718-727 (1973).
- [MAC75] MacDougall, M. H., "Process and Event Control in ASPOL", Proc. Symp. on the Simulation of Computer Systems, Boulder, Colorado, August 1975, 35-51.
- [PET77] Peterson, J. L., "Petri Nets", Computing Surveys, 9, 223-252 (1977).
- [SAN79] Sanguinetti, J., "A Technique for Integrating Simulation and System Design", Proc. of Conf. on Simulation, Measurement and Modeling of Computer Systems, Boulder, Colorado, August 1979, 163-172.
- [SHA79] Shaw, M., "A Formal System for Specifying and Verifying Program Performance", (Dept. of Computer Science Report, Carnegie-Mellon University), June 1979.
- [SMI79a] Smith, C. and Browne, J. C., "Performance Specifications and Analysis of Software Designs", Proc. of Conf. on Simulation, Measurement and Modeling of Computer Systems, Boulder, Colorado, August 1979, 173-182.
- [SMI79b] Smith, C. and Browne, J. C., "Modeling Software Systems for Performance Predictions", Proc. CMG, Group X, Dallas, Texas, December 1979.
- [SMI80] Smith, C. and Browne, J. C., "Aspects of Software Design Analysis: Concurrency and Blocking" (to be presented at Performance '80, Toronto, May 1980).
- [SPI77] Spirn, J. R., "Program Behavior: Models and Measurements", (Elsevier North-Holland, Ltd., New York, 1977).

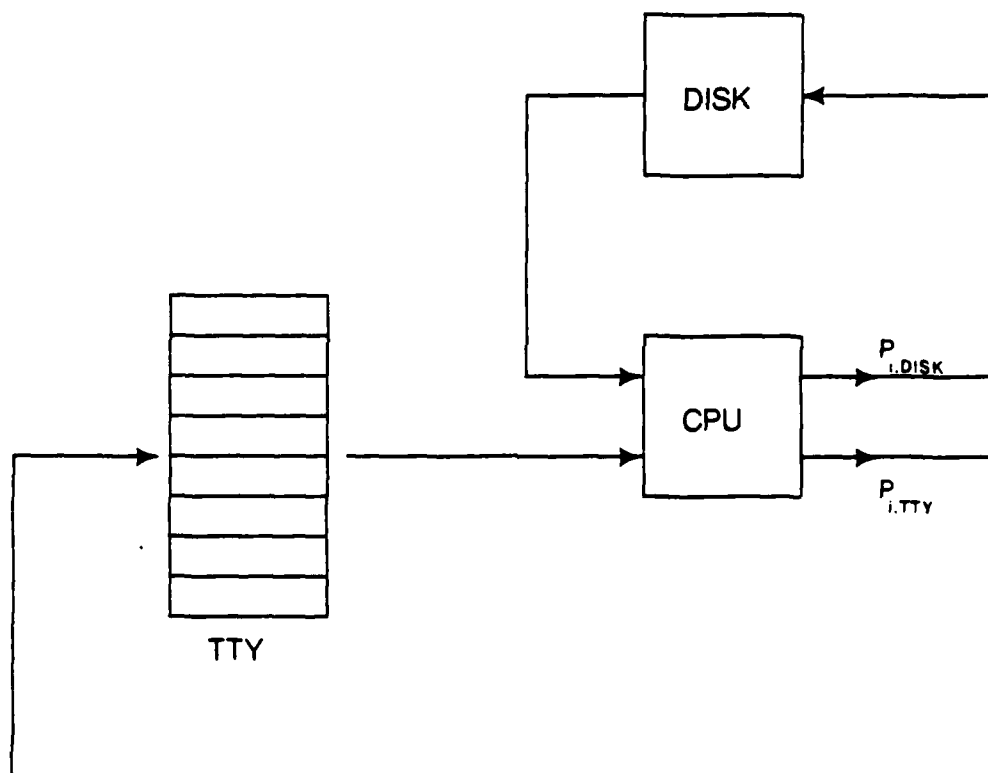


Figure 1

Simple Queueing Network Model of a Computer System

STATISTICAL MEASURES OF SOFTWARE RELIABILITY

Richard A. DeMillo
Georgia Institute of Technology

Frederick G. Sayward
Yale University

DRAFT COPY - ONR Study Group on Software Metrics - DO NOT CIRCULATE

I. INTRODUCTION

Estimating program reliability presents many of the same problems as measuring software performance and cost: the central technical issue concerns the existence of an independent objective scale upon which may be based a qualitative judgement of the ability of a given program to function as intended in a specified environment over a specified time interval. Several scales have already been proposed. For example, a program may be judged reliable if it has been formally proved correct [1], if it has been run against a valid and reliable test data set [2], or if it has been developed according to a special discipline [3]. While these concepts may have independent interest, they fail to capture the most significant aspect of reliability estimation as it applies to software: Most software is unreliable by these standards, but the degree of unreliability is not addressed. A useful program which has not been proved correct is unreliable, but so is, say, the null program (unless by some perversity of specification the null program satisfies the designer); an operationally meaningful scale of reliability should distinguish these extremes.

What is needed is a measure $R(t)$ which, for a given piece of software (i.e., a system, subsystem, program, or program module) gives an index of operational reliability in the time interval $[0, t]$. The most commonly proposed such index is the reliability function of traditional reliability theory [4,5,6]:

$R(t)$ = probability of survival at time t .

In the sequel, we will sketch the outlines of the traditional theory that is most relevant to software reliability estimation, give a brief critical analysis of the use of the traditional theory in measuring reliability, and propose a new use of the $R(t)$ measure which we believe more closely fits the intuitive requirements of the scale we asked for above.

2. THE STATISTICAL THEORY

$R(t)$ is to be interpreted as the probability of satisfactory performance of the system in the time interval $[0, t]$. It is an underlying assumption that satisfactory performance at time t implies satisfactory performance in the interval $[0, t]$. A second assumption is that the form of the theory does not change from system to system; in particular, it should not matter whether one estimates $R(t)$ for a total system, a subsystem, or a single component.

The second assumption suggests the following analysis of complex systems. Let R_i be the reliability function for the i th component of the system and define a random variable x_i for each component as follows.

$$x_i = \begin{cases} 1, & \text{if } R_i(t) \geq a_i \\ 0, & \text{otherwise} \end{cases}$$

The performance of the system is then determined by a 0-1 valued function

$$\phi(x_1, \dots, x_n).$$

Simple examples of such performance functions are the functions for series systems,

$$\phi(x_1, \dots, x_n) = x_1 x_2 \dots x_n = \min\{x_i | 1 \leq i \leq n\},$$

and for "parallel" systems,

$$\phi(x_1, \dots, x_n) = 1 - (1 - x_1) \dots (1 - x_n) = \max\{x_i | 1 \leq i \leq n\}.$$

For more complex situations, one may place additional requirements on the performance function; for instance, the function may have to be coherent:

$$\text{for all } i, x_i \leq y_i \implies \phi(y_1, \dots, y_n) \geq \phi(x_1, \dots, x_n).$$

The reliability function for the entire system is then

$$R = \text{Prob}\{\phi(x_1, \dots, x_n) = 1\}.$$

For extremely simple models of subsystem failure (e.g., fixed independent probabilities of satisfactory performance) such an analysis gives the combinatorial probabilities of success/failure in a very handy form.

For realistically complex systems, however, $R(t)$ is determined by probability distributions which draw their properties from observable parameters of the system. For computer programs, as well as many other systems, it is inconvenient to estimate satisfactory performance directly. Instead one uses the observed failures of the system. Let $F(t)$ be the failure distribution of the system and let f be the corresponding probability density function. Then

$$R(t) = 1 - F(t) = \int_t^{\infty} f(y) dy.$$

Often, the reliability distribution is more conveniently expressed in terms of the failure rate (or hazard rate). Let $B(t, h)$ represent the conditional probability of failure by time $t+h$ given survival at time t :

$$r(t) = \lim_{h \rightarrow 0} \{B(t, h)/h\} = f(t)/R(t)$$

The failure rate determines the reliability since

$$1 - F(t) = R(t) = \exp\left(-\int_0^t r(y) dy\right).$$

One obtains a reliability function for a given system by a variety of paths: nontechnical considerations like mathematical tractability, empirical observations, or theoretical analysis of more fundamental properties of the system which is subject to failure.

The exponential distribution. The exponential distribution is exactly characterized by those systems which have constant failure rate.

$$R(t) = \exp(-\theta t)$$

$$r(t) = \theta \exp(-\theta t) / \exp(-\theta t) = \theta.$$

The exponential distribution is the most widely used of the reliability models, even when there is slight evidence to justify its use. It is, of course, among the most tractable of the statistical models, but it can lead to serious errors when the underlying distribution corresponds to failures that do not occur randomly or which depend on the past history of the system. The popularity of the exponential distribution (and the tendency to observe it in complex systems) is no doubt due in part to the following fact. Let R_i be the reliability of the i th component and let $R = \prod R_i$. If there are N components in the system, define $\tilde{r}(N)$ to be $r_1(0) + \dots + r_N(0)$. Then R tends to be an exponential distribution:

$$R(t) \rightarrow \exp(-\tilde{r}(N)t) \text{ as } N \rightarrow \infty,$$

whenever three technical conditions are met.

1. each failure rate grows as

$$r_i(t) = r_i(0) + a_i t^\theta$$

as $t \rightarrow 0$.

2. $\tilde{r}(N) \rightarrow \infty$ as $N \rightarrow \infty$.

3. $\tilde{r}(N)(a_1 + \dots + a_N)$ is bounded by a fixed constant.

These conditions are however not easy to satisfy, and several of the most common distributions of reliability theory fail one or more of the restrictions.

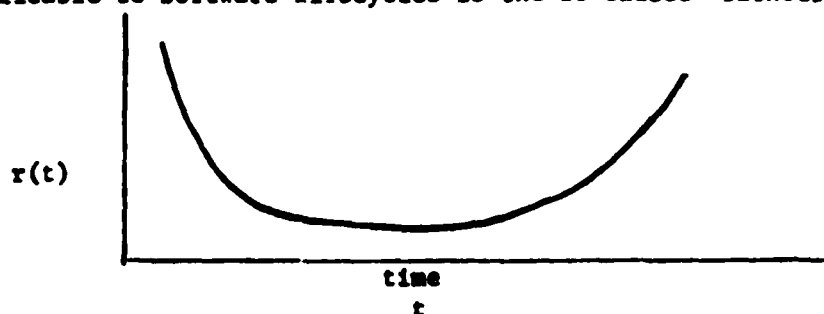
The Weibull distribution. In an attempt to circumvent the more serious deficiencies of the exponential distribution, one is led to a family of distributions in which the failure rate is a function of time.

$$R(t) = \exp(-t^a/\theta),$$

$$r(t) = at^{a-1}/\theta.$$

Obviously, the failure rate is increasing or decreasing depending on the value of a . The properties of the $R(t)$ distribution for certain choices of a and θ make it a good descriptor of mechanical "breakdown" phenomena, such as failures due to metal fatigue or other failures due to stress. The Weibull distribution varies wildly from the exponential distribution in its statistical properties (compare, for example, the n th moments of both) and thus cannot be used interchangeably with predictable results. A special case of the Weibull distribution called the Rayleigh distribution is common in software reliability studies. This class of distributions fails to satisfy restrictions (1) and (3) above, so that it is difficult to argue that interconnections of Rayleigh-distributed components may behave with simpler failure rates.

Truncated normal distribution. The failure rate that is most commonly cited as applicable to software lifecycles is the so-called "bathtub" curve



As opposed to the Weibull-type distributions, the normal distributions model systems which "wear out", e.g., light bulbs. Rather than dealing directly with the reliability functions which may only be applicable to a portion of the system life cycle, it is often convenient to combine several failure rates to model modes of failure which apply to specific stages of development. The wear-out models are particularly attractive in this regard. During the initial stages of operation of a large number of copies of identical systems, there is a high incidence of immediate failures ("infant mortality"). This initiation process is followed by a relatively stable steady-state region until, near the end of the useful lifetime of the systems, the incidence of failures rises.

If $\phi(x)$ is the probability density function for a normal distribution and $\Phi(x)$ is the corresponding distribution function, then even though the expression for $R(t)$ involves an integral that cannot be expressed in closed form, the failure rate is easily expressed:

$$r(t) = a\phi(at-t_0)/\Phi(at_0).$$

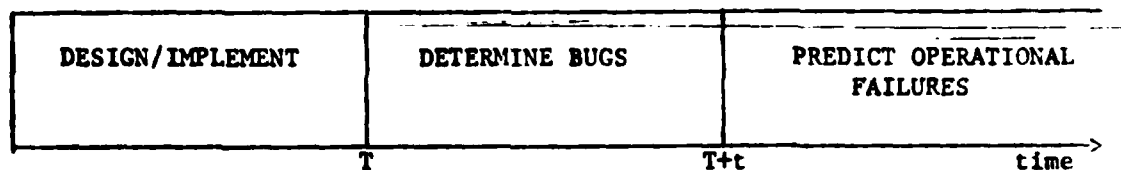
Renewal theory. A common application for the reliability theory sketched above is in the prediction of cost of operation as the system functions within some prescribed maintenance policy. By knowing statistical properties of operational periods (e.g., such statistics as mean time between failure) and repair periods (e.g., replacing a failed component reduces the number of failed components by one) a variety of useful system parameters can be calculated. In concert with system design and maintenance policies (for example, survivability may be insured by redundancy and repair effort reduced by keeping a stock of spare parts) standard optimization models can be used to minimize total operating costs due to failure. The mathematics

of the optimization techniques dictates very simple statistical models for these applications. Among the simplifications which seem to be essential one frequently finds:

1. failures are immediately detectable and attributable to a single component,
2. during periods of reduced operation the remaining load is shared by the rest of the system,
3. the renewed system and its predecessor have identically distributed failures.

3. SOFTWARE MODELS

The descriptions of extant software models presented here are obtained from the DACS summaries [7]. Although a wide variety of models are covered in the DACS extracts, we will highlight those which fall within the time-dependent reliability models of the traditional theory. The following developmental chart illustrates the use of these models.



At time T in the lifecycle of the software system, the statistical model is determined, usually by controlled observations of system errors (the "failures" of the traditional theory). In the interval $[T, T+t]$ enough data is to be gathered to determine a failure rate $r(t)$. This coupled with a variety of subjective evidence concerning the behavior of the system in operation then determines an estimate of the underlying failure distribution. The predictive model is then applied in the interval $[T+t, T+t+k]$,

where k is a "regenerative" time less than the operational lifetime of the system.

The following table summarizes the utilization of the traditional models summarized in [7].

Distribution	Number of Models
Exponential (incl. geometric and Poisson)	7
Rayleigh or Weibull	3
Normal	1

As described above, the Exponential-Weibull distributions describe system failures when the system components are stressed materials. While there may be a body of experimental data which fits the appropriate failure rate functions, the assumption of one of these distributions as the underlying reliability distribution forces one or more of a number of questionable assumptions concerning errors in software. For example:

1. The number of initial program errors can be reliably estimated.
2. Error detection rate (failure rate) is proportional to the number of remaining errors.
3. Errors are discovered one at a time.
4. Once an error is detected, it can be located and removed immediately.
5. Error occurrence rate is constant.
6. Removing an error reduces the total number of errors by one.
7. Error occurrences are statistically independent.
8. The distribution of program inputs is known.
9. Error detection rate is proportional to the debugging effort.
10. Program size is constant over the lifetime of the program.

The experimental evidence to support these assumptions is contradictory [8]. Common sense would suggest that statistical models intended to describe physical systems would be ill-adapted to software, but it is still possible that nature allows an aggregate description of the error occurrence rate in typical software that is useful in practice. One test of this is careful experimentation on the assumptions 1-10. Allen Acree of the Georgia Tech Testing Laboratory has, for instance, gathered extensive data [9] on failure rates as related to the number of remaining errors in small programs (50-1000 lines) and found that it does support the exponential rate. In other studies by Acree and Timothy Budd of the Yale Testing Laboratory, however, [9,10] there is rather clear evidence that the strategic assumptions of independence of errors, single error occurrence, and immediate removal fail drastically in even moderately large systems. There is also considerable evidence that for large systems, most remaining errors lie in unexecuted portions of code, which means that failure rates cannot depend on either number of remaining errors or debugging effort for these systems.

Certain of the remaining assumptions also appear hard to confirm. For example, there is no methodologically acceptable way of estimated initial error percentages. The most commonly proposed technique is the error seeding procedure described in [11], which is based on the population statistics calculations described in Feller's classic text [12]. While this procedure gives the best unbiased estimator for certain random populations, it can badly overestimate or underestimate the number of errors when there is nonrandom mixing of seeded and natural errors. There is experimental evidence that such nonrandom mixing must take place [9,10].

Finally, the use of the Rayleigh or Weibull distributions to model program modules presents severe mathematical difficulties for macro-modelling, since these distributions are nonreproductive; that is, overall system description is not a simple aggregate of module descriptions.

The usual validation of these models is the posterior observation of failure rates and costs. The experimental technique is deficient and lacks some essential size constraints to allow statistical sampling techniques to be used with acceptable confidence.

If, indeed, the classical theory can be imported to software reliability there must be considerable attention paid to deriving relevant distributions from first principles and to developing the appropriate experimental and data gathering [12] techniques. It should be noted, however, that software errors must be viewed as design errors [8] (since there is no material wearing or stressing) and classical reliability theory does not deal satisfactorily with design errors.

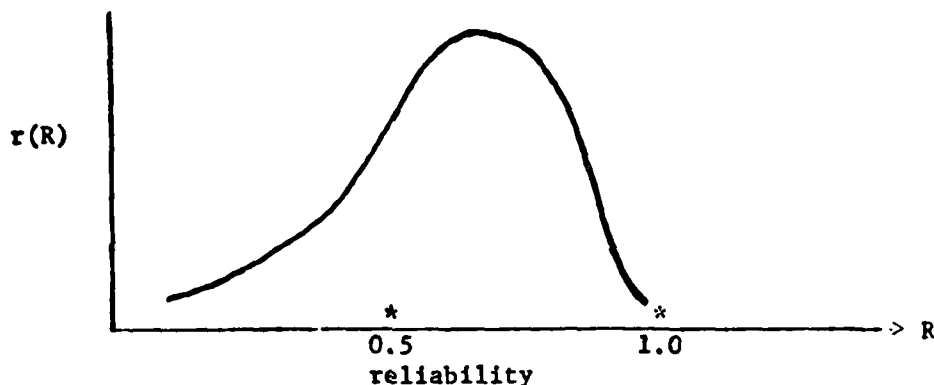
4. CONFIDENCE ESTIMATION

In a sense, the goals of traditional reliability theory and software reliability measurement are fundamentally at odds with each other. We have little opportunity to create large populations of identical systems and observe failure rates to obtain statistically meaningful measurements of failure distributions as indications of relative frequencies.

The more common situation is that a fixed programming-testing-validation method is imposed with a certain confidence level. The actual reliability of the system is then intertwined with this estimate of the system's reliability (since we stop testing when our estimate is satisfied).

We should then ask for experiments to confirm the level of confidence

in the chosen methodology. In traditional statistical approaches to reliability, such objectively obtained parameters as number of error observed per unit time are used to infer the appropriate distributions. We suggest that the observable parameters represent instead a varying quantity which represents a prior estimate of reliability. That is, we use a function $r(R)$ to represent our level of confidence in the reliability level R of the software. For example, a program that has been subjected to mutation analysis [13] might be described by a function r with the shape shown below.



It is important to note that $r(R)$ is merely a quantitatively assessed judgement; it is not necessarily an objective probability obtained by classical techniques. The r function may, however, be based on solid evidence and may form the basis for a scientific hypothesis concerning the long run behavior of a system. By applying Bayes' theorem [11], it is possible to gather "hard" evidence to support or deny the estimate of confidence $r(R)$. From now on, we treat R as a random variable with probability density function $r(R)$.

Suppose that in an experiment designed to refute a given reliability level R we make a large number of observations and gather a statistic, say,

x , which is distributed as $E(x|R)$ for a given level R . Think of E as a distribution of errors for a given reliability; this is a quantity which can be observed and inferred in a given programming environment. The joint distribution for x and R is $F(x,R)$. Let the associated pdf's be e and f , so that

$$f(x,R) = r(R)e(x|R),$$

the marginal density for x is:

$$g(x) = \int_R r(R)e(x|R)dR.$$

Then the conditional distribution for R is given by the density

$$h(R|x) = r(R)e(x|R)/g(x),$$

which represents our level of confidence in R given the results of the experiment yielding x .

Even though these results merely restate Bayes' theorem, the interpretation implied by the results of the experiment yielding x is not necessarily subjective. The key is rather in the degree of objectivity with which $r(R)$ is obtained.*

At the heart of this treatment of software reliability is a new view of the role of statistical statements. This point of view rejects the idea that a meaningful probability of correct operation can be assigned to a piece of software. Rather, any such assessment must be interpreted as a "level of confidence" in the process used to validate the program. This effectively shifts the statistical burden from the program to the methodology used to produce and validate it. One problem with the traditional approach, which

*cf. Peller's criticism of Bayesian analysis in reliability [12, p. 124].

we noted briefly above, is that there is no acceptable sense in which the frequency interpretation of the reliability function $R(t)$ can apply to software; there is, for example, no reasonable application of the law of large numbers to obtain failure rates. This view of probability -- the limit of a physically observable relative frequency -- is not the only view. It is possible to construct alternative subjective probabilities. In fact, James Bernoulli in 1713 described probabilities as "degrees of confidence."

Even though we use the term "subjective" in describing the non-frequency version of $R(t)$, it does not follow that the measure represents an ad hoc assessment of reliability. For example, the prior distribution may not really represent a subjective estimate of reliability. It may be the result of experimental research and summarization. This suggests an iterative procedure for "updating" the state of knowledge regarding the reliability of the validation methodology. In other words, an assessment of reliability represents a judgement relative to a prior reference "standard" which the user presumably understands, believes, and against which he is being required to wager. Hence, for the reliability measurement to be useful, it must only order uncertain events so that the user may consistently win when he adopts the rational betting strategy.

REFERENCES

- [1] Z. Manna, Mathematical Theory of Computation, McGraw Hill, 1974.
- [2] S. Gerhart and J. Goodenough, "Toward a Theory of Test Data Selection," in R. Yeh (Editor), Current Trends in Programming Methodology: Vol. 2, Program Validation, Prentice Hall, 1977, pp. 44-79.
- [3] E. Dijkstra, A Discipline of Programming, Prentice Hall, 1977.
- [4] K.C. Kapur and L.R. Lamberton, Reliability in Engineering Design, Wiley, 1977.

- [5] M. Zelen (Editor), Statistical Theory of Reliability, University of Wisconsin Press, 1964.
 - [6] R.E. Barlow and F. Proschan, Mathematical Theory of Reliability, Wiley, 1965.
 - [7] DACS, Quantitative Software Models, March 1979.
 - [8] R. Longbottom, Computer System Reliability, Wiley, 1980.
 - [9] A.T. Acree, "On Mutation," PhD Thesis, School of Information and Computer Science, Georgia Institute of Technology, June, 1980.
 - [10] T.A. Budd, "Mutation Analysis of Program Test Data," PhD Thesis, Department of Computer Science, Yale University, June, 1980.
 - [11] T. Gilb, Software Metrics, Prentice Hall, 1978.
 - [12] W. Feller, An Introduction to Probability Theory and Its Applications, Vol. 1, Wiley, 1968.
 - [13] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Mutation Analysis," Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.
-

THE MEASUREMENT OF SOFTWARE QUALITY AND COMPLEXITY

Bill Curtis
Information Systems Programs
General Electric Company
Arlington, Virginia

Uses for Software Metrics

The measurement of software complexity is one facet of a larger effort to measure important software characteristics. Measurements of software characteristics can provide valuable information throughout the software life cycle. During development, measurements can be used to predict the resources which will be required in future phases of the project. For instance, metrics developed from the detailed design can be used to predict the amount of effort that will be required to implement and test the code. Metrics developed from the code can be used to predict the number of errors that may be found in subsequent testing or the difficulty involved in modifying a section of code. Because of their potential predictive value, software metrics can be used in at least three ways:

1. Management information tools - As a management tool, metrics provide several types of information. First, they can be used to predict future outcomes as discussed above. Measurements can be developed for costing and sizing at the project level, such as in the models proposed by Freiman and Park (1980), Putnam (1978), and Wolverton (1974). Other models have been developed for estimating productivity by Freberger and Basili (1979) and Walston and Felix (1977). Such metrics allow managers to assess progress, future problems, and resource requirements. If these metrics can be proven reliable and valid indicators of development processes, they provide an excellent source of management visibility into a software project.
2. Measures of software quality - Interest grows in creating quantifiable criteria against which a software product can be judged (Mohanty, 1979). An example criterion would be the minimally acceptable mean-time-between-failures. These criteria could be used as either acceptance standards by a software acquisition manager or as guidance to potential problems in the code during software validation and verification (Walters, 1979).
3. Feedback to software personnel - Elshoff (1978) has used a software complexity metric to provide feedback to programmers about their code. When a section grows too complex they are instructed to redesign the code until metric values are brought within acceptable limits.

The three uses described above suggest a difference between measures of process and product. Measures of process would include the resource estimation metrics described as potential management tools. Measures of cost and productivity quantify attributes of the development process. However, they convey little information about the actual state of the software product. Measures of the product represent software characteristics as they exist as a given time, but do not indicate how the software has evolved into this state. Measures used for feedback to programmers or as quality criteria fall within this second category.

Belady (1977) argues that it will be difficult to develop a metric which can represent both process and product. Development of such a metric or set of metrics will require a model of how software evolves from a set of requirements into an operational program. Charting the sequential phases of the software life cycle will not provide a sufficient model. Some progress is being made on system evolution by Lehman and his colleagues at Imperial College in London (Benyon-Tinker, 1980; Lehman, 1980). In the remainder of this section, I will deal with measures of product rather than process.

Omnibus Approaches to Quantifying Software

There have been several attempts to quantify the elusive concept of software quality by developing an arsenal of metrics which quantify numerous factors underlying the concept. The most well-known of these metric systems are those developed by Boehm, Brown, Kaspar, Lipow, MacLeod, and Meritt (1978), Gilb (1977), and McCall, Richards, and Walters (1977). The Boehm et al. and McCall et al. approaches are similar, although differing in some of the constructs and metrics they propose. Both of these systems have been developed from an intuitive clustering of software characteristics (Figure 1).

The higher level constructs in each system represent 1) the current behavior of the software, 2) the ease of changing the software, and 3) the ease of converting or interfacing the system. From these primary concerns Boehm et al. develop seven intermediate constructs, while McCall et al. identify eleven quality factors. Beneath this second level Boehm et al. create twelve primitive constructs and McCall et al. define 23 criteria. For instance, at the level of a primitive construct or criterion both Boehm et al. and McCall et al. define a construct labeled "self-descriptiveness". For Boehm et al. this construct underlies the intermediate constructs of testability and understandability, both of which serve the primary use of measuring maintainability. For McCall et al. self-descriptiveness underlies a number of factors included under the domains of product revision and transition.

Primitive constructs and criteria are operationally defined by sets of metrics which provide the guidelines for collecting empirical data. The McCall et al. system defines 41 metrics consisting of 175 specific elements. Thus, the metrics themselves represent composites of more elementary measures. This proliferation of measures should ultimately be reduced to a

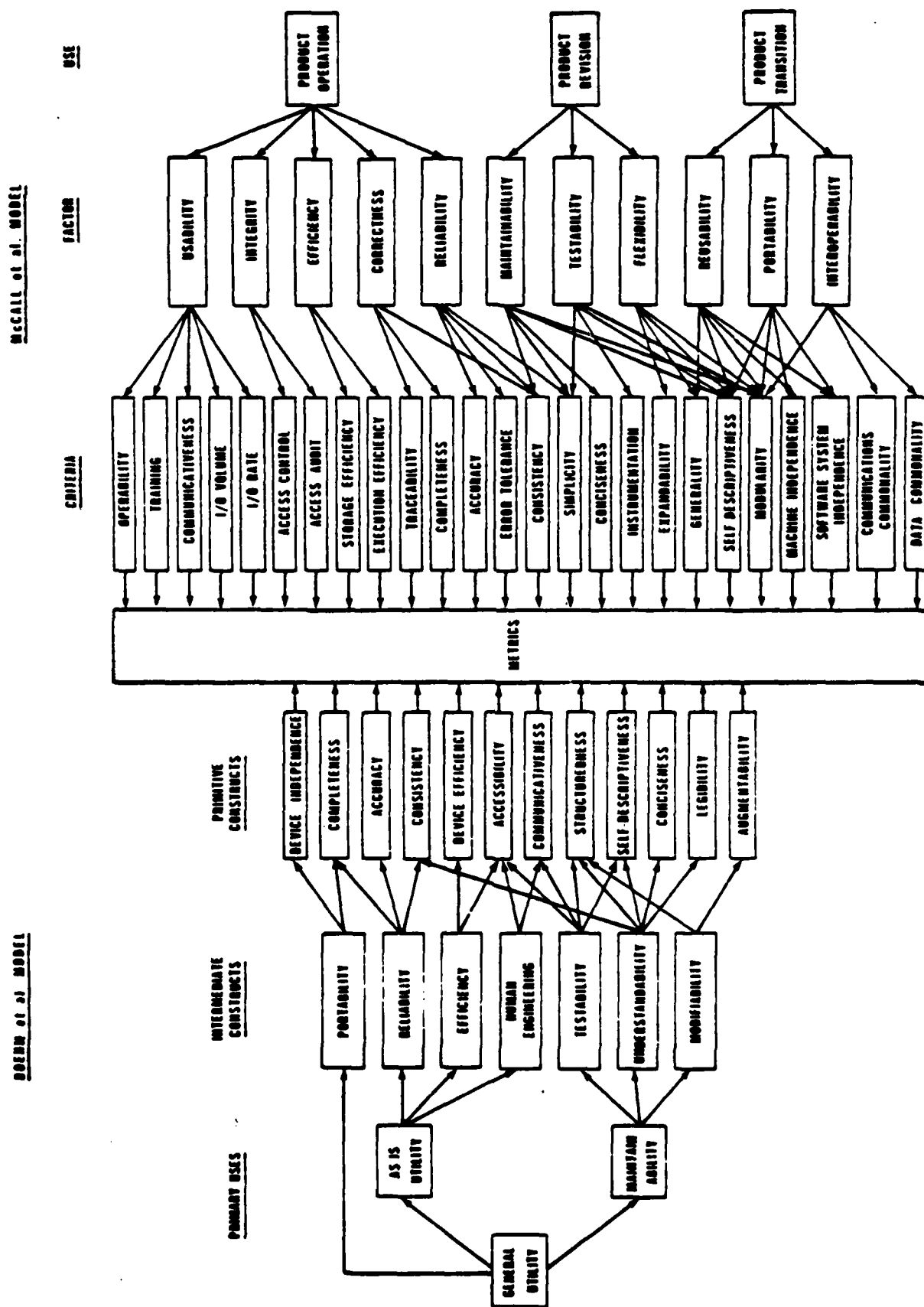


Figure 1. The Boehm et al. and McCall et al. software quality models.

manageable set which can be automated. Reducing their number will require an empirical evaluation of which metrics carry the most information and how they cluster. There are a number of multivariate statistical techniques available for such analyses (Morrison, 1967).

No software project can stay within a reasonable budget and maximize all of the quality factors. The nature of the system under development will determine the proper weighting of quality factors to be achieved in the delivered software. For instance, reliability was a critical concern for Apollo space flight software where human life was constantly at risk. For business systems, however, maintainability is typically of primary importance. In many real-time systems where space or time constraints are critical, efficiency takes precedence. However, optimizing code often lowers its quality as indexed by other factors such as maintainability and portability. Figure 2 presents a tradeoff analysis among quality factors performed by McCall et al. (1977).

The omnibus approach to metric development had its birth in the need for measures of software quality, particularly during system acquisition. However, the development of these metrics has not spawned explanatory theory concerning the processes affected by software characteristics. The value of these metric systems in focusing attention on quality issues is substantial. However, there is still a greater need for quantitative measures which emerge from the modeling of software phenomena. Much of the modeling of software characteristics has been performed in an attempt to understand software complexity.

Software Complexity

The measurement of software complexity is receiving increased attention, since software accounts for a growing proportion of total computer system costs. Complexity has been a loosely defined term, and neither Boehm et al. nor McCall et al. included it among their constructs of software quality. Complexity is often considered synonymous with understandability or maintainability.

Two separate focuses have emerged in studying software complexity: computational and psychological complexity. Computational complexity relies on the formal mathematical analysis of such problems as algorithm efficiency and use of machine resources. Rabin (1977) defines this branch of complexity as "the quantitative aspects of the solutions to computational problems" (p. 625). This topic is discussed in a later paper in this collection by James Browne and William Lynch. In contrast to this formal analysis, the empirical study of psychological complexity has emerged from the understanding that software development and maintenance are largely human activities. Psychological complexity is concerned with the characteristics of software which affect programmer performance.

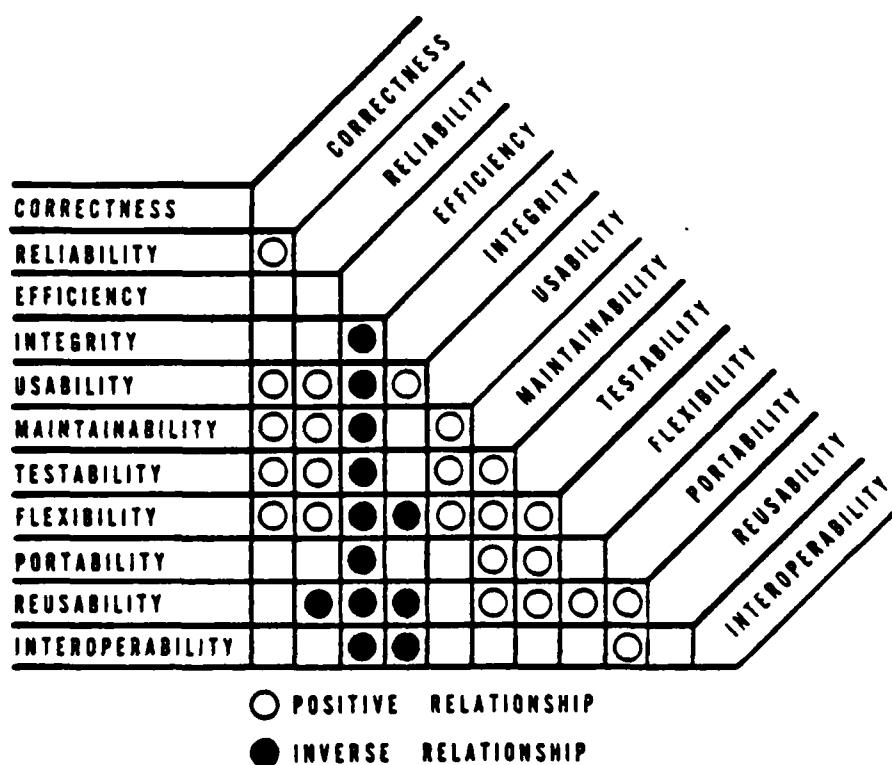


Figure 2. McCall et als.' tradeoff analysis among software quality factors.

The investigation of computational and psychological complexity has been carried on without a unifying definition for the construct of software complexity. There do, however, seem to be common threads running through the complexity literature which suggest the following definition:

Complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software. (Curtis, 1980, p.102)

Several important points are implied by this definition. First, the focus of complexity is not merely on the software, but on the software's interactions with other systems. Complexity has little meaning in a vacuum, it requires a point of reference. This reference takes meaning only when developed from other systems such as machines, people, other software packages, etc. It is these systems that are affected by the "complexity" of a piece of software. Worrying about software characteristics in the absence of other systems has merit only in an artistic sense, and measures of "artistic" software are quite arbitrary. However, when there is an external reference (criterion) against which to compare software characteristics, it becomes possible to operationally define complexity.

Second, explicit criteria are not specified. This definition allows mathematicians and psychologists to become strange bedfellows since it does not specify the particular phenomena to be studied. Rather, this definition steps back a level of abstraction and describes the goal of complexity research and the reference against which complexity takes meaning. Complexity is an abstract construct, and operational definitions only capture specific aspects of it.

The second point suggests the third: complexity will have different operational definitions depending on the criterion under study. Operational definitions of complexity must be expressed in terms which are relevant to processes performed in other systems. Complexity is defined as a property of the software interface which affects the interaction between the software and another system. To assess this interaction, we must quantify software characteristics which are relevant to it. A model of software complexity implies not only a quantification of software characteristics, but also a theory of processes in other systems. Thus, the starting point for developing a metric is not an ingenious parsing of software characteristics, but an understanding of how other systems function when they interact with software.

The following steps should be followed in modeling an aspect of software complexity:

- 1) Define (and quantify) the criterion the metric will be developed to predict.
- 2) Develop a model of processes in the interacting system which will affect this criterion.

- 3) Identify the properties of software which affect the operation of these processes.
- 4) Quantify these software characteristics.
- 5) Validate this model with empirical research.

The importance of this last point cannot be overemphasized. Nice theories become even nicer when they work. Preparing for the rigors of empirical evaluation will probably result in fewer metrics and tighter theories. Results from validation studies make excellent report cards on the current state-of-the-art.

Belady (1980) has categorized much of the existing software complexity literature. First, he distinguishes different software characteristics which are measured as an index of complexity: algorithms, control structures, data, or composites of structures and data. In a second dimension he describes the type of measurement employed: informal concept, construct counts, probabilistic/statistical treatments, or relationships extracted from empirical data. Most research has concerned counts of software characteristics, particularly control structures and composites of control structures and data. I will review some of the complexity research in these two areas and compare them to a system level metric.

Control Structures

A number of metrics having a theoretical base in graph theory have been proposed to measure software complexity by assessing the control flow (Bell & Sullivan, 1974; Chen, 1978; Green, Schneidewind, Howard, & Pariseau, 1977; McCabe, 1976; Richards & Chang, 1975; Woodward, Hennell, & Hedley, 1979). Such metrics typically index the number of branches or paths created by the conditional expressions within a program. McCabe's metric will be described as an example of this approach since it has received the most empirical attention.

McCabe (1976) defined complexity in relation to the decision structure of a program. He attempted to assess complexity as it affects the testability and reliability of a module. McCabe's complexity metric, $v(G)$, is the classical graph-theory cyclomatic number indicating the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. When combined these paths generate the complete control structure of the program. McCabe's $v(G)$ can be computed as the number of predicate nodes plus 1, where a predicate node represents a decision point in the program. It can also be computed as the number of regions in a planar graph (a graph in regional form) of the control flow. This latter method is demonstrated in Figure 3.

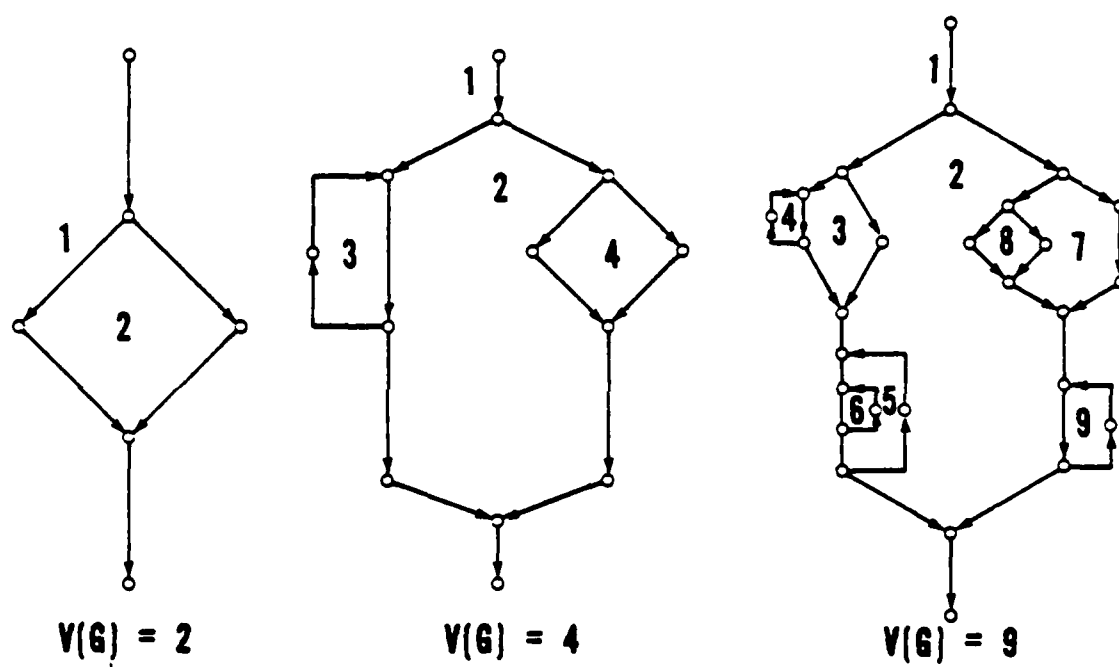


Figure 3. Computation of McCabe's $v(G)$.

McCabe argues that his metric assesses the difficulty of testing a program, since it is a representation of the control paths which must be exercised during testing. From experience he believes that testing and reliability will become greater problems in a section of code whose $v(G)$ exceeds 10.

Basili and Reiter (1980) and Myers (1977) have developed different counting methods for computing cyclomatic complexity. These differences involved counting rules for CASE statements and compound predicates. Definitive data on the most effective counting rules have yet to be presented. Nevertheless, considering alternative counting schemes to those originally posed by the author of a metric is important in refining measurement techniques.

Evidence continues to mount that metrics developed from graphs of the control flow are related to important criteria such as the number of errors existing in a segment of code and the time to find and repair such errors (Curtis, Sheppard, & Milliman, 1979; Feuer & Folkes, 1979; Schneidewind & Hoffman, 1979). Chen (1978) developed a variation of the cyclomatic number which indexed the nesting of IF statements and related this to the information-theoretic notion of entropy within the control flow. He reported data from eight programmers indicating that productivity decreased as the value of his metric computed on their programs increased. Thus, the number of control paths appears directly or indirectly related to psychological complexity.

Software Science

The best known and most thoroughly studied of what Belady (1980) classifies as composite measures of complexity has emerged from Halstead's theory of Software Science (1972, 1977). In 1972, Maurice Halstead argued that algorithms have measurable characteristics analogous to physical laws. Halstead proposed that a number of useful measures could be derived from simple counts of distinct operators and operands and the total frequencies of operators and operands. From these four quantities Halstead developed measures for the overall program length, potential smallest volume of an algorithm, actual volume of an algorithm in a particular language, program level (the difficulty of understanding a program), language level (a constant for a given language), programming effort (number of mental discriminations required to generate a program), program development time, and number of delivered bugs in a system. Two of the most frequently studied measures are calculated as follows:

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2^{\eta_2}}$$

where V is volume, E is effort, and

η_1 = number of unique operators

η_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

Halstead's theory has been the subject of considerable evaluative research (Fitzsimmons & Love, 1978). Correlations often greater than .90 have been reported between Halstead's metrics and such measures as the number of bugs in a program (Bell & Sullivan, 1974; Cornell & Halstead, 1976; Fitzsimmons, 1978; Funami & Halstead, 1976; Ottenstein, 1979), programming time (Gordon & Halstead, 1976; Sheppard, Milliman & Curtis, 1979), debugging time (Curtis, Sheppard & Milliman, 1979; Love & Bowman, 1976), and algorithm purity (Bulut & Halstead, 1974; Elshoff, 1976; Halstead, 1973).

My colleagues and I have evaluated the Halstead and McCabe metrics in a series of four experiments with professional programmers. In the first two experiments (Curtis, Sheppard, Milliman, Borst, & Love, 1979) problems in the experimental procedures, a limit on the size of programs studied, and substantial differences in performance among the 36 programmers involved in each suppressed relationships between the metrics and task performance. In fact it did not appear that the metrics were any better than the number of lines of code for predicting performance. However, in the third experiment (Curtis, Sheppard, & Milliman, 1979) we used longer programs, increased the number of participants to 54, and eliminated earlier procedural problems. We found both the Halstead and McCabe metrics superior to lines of code for predicting the time to find and fix an error in the program.

In the final experiment (Sheppard, Milliman, & Curtis, 1979), we asked nine programmers to each create three simple programs (e.g., find the maximum and minimum of a list of numbers) from a common specification of each program. The best predictor of the time required to develop and successfully run the program was Halstead's metric for program volume (Figure 4). This relationship was slightly stronger than that for McCabe's $v(G)$, while lines of code exhibited no relationship.

The datapoints circled in Figure 4 represent the data from a program whose specifications were less complete than those of the other two programs studied. The prediction of development time for this program was poor. We have observed in other studies that outcomes are more predictable on projects where a greater discipline regarding software standards and practices was observed (Milliman & Curtis, 1979, 1980). This experiment suggests that better prediction of outcomes may occur when more disciplined software development practices (e.g., more detailed program specifications) reduce the often dramatic performance differences among programmers.

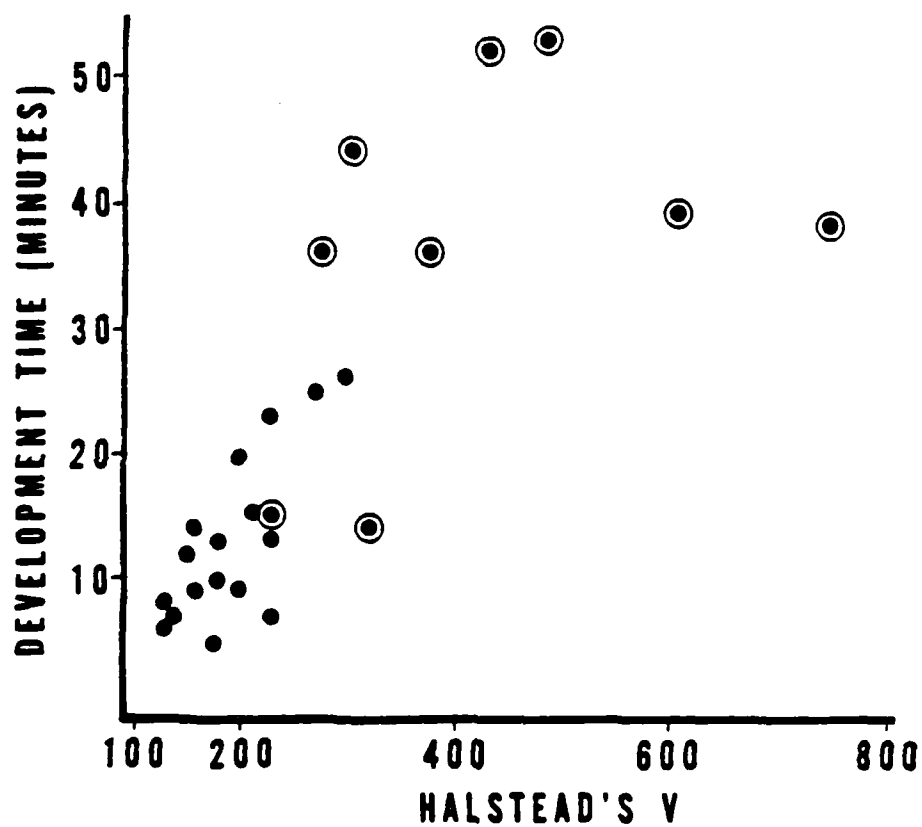


Figure 4. Scatterplot of Halstead's V against development time from Sheppard et al.

AD-A087 412

YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE

F/G 9/2

DRAFT SOFTWARE METRICS PANELS FINAL REPORT. PAPERS PRESENTED AT--ETC(U)

JUN 80 A J PERLIS, F G SAYWARD, M SHAW

N00014-79-C-0672

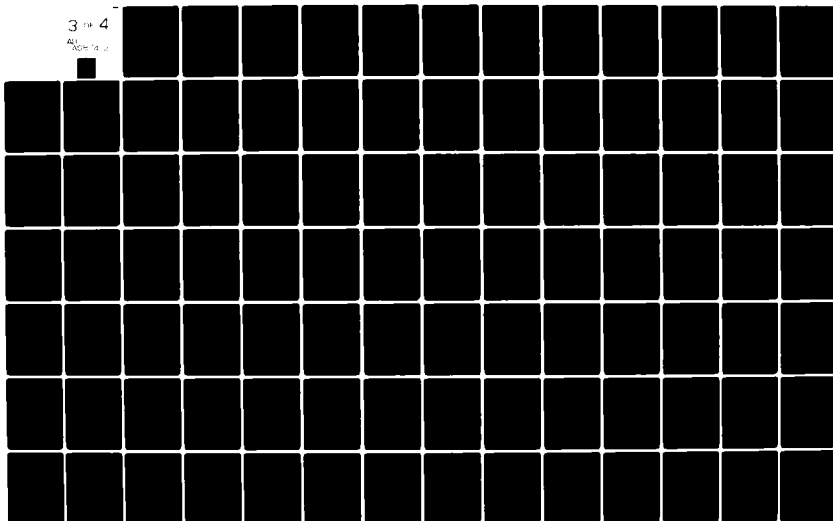
UNCLASSIFIED

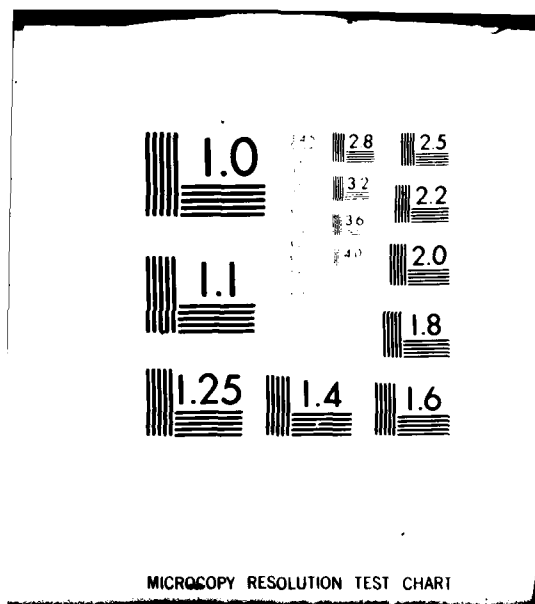
RR-182/80

NL

3 of 4

ALL INFORMATION CONTAINED
HEREIN IS UNCLASSIFIED





In these experiments we found Halstead's and McCabe's metrics to be valid measures of psychological complexity, regardless of whether the program they were computed on was developed by the programmer under study or by someone else. We concluded that there is considerable promise in using complexity metrics to predict the difficulty programmers will experience in working with software. Similar conclusions have been reached by Baker and Zweben (1979) on an analytical rather than empirical evaluation of the Halstead and McCabe metrics.

Halstead's metrics have proven useful in actual practice. For instance, Elshoff (1978) has used these metrics as feedback to programmers during development to indicate the complexity of their code. When metric values for their modules exceed a certain limit, programmers are instructed to consider ways of reducing module complexity. Bell and Sullivan (1974) suggest that a reasonable limit on the Halstead value for length is 260, since they found that published algorithms with values above this figure typically contained an error.

Regardless of the empirical support for many of Halstead's predictions, the theoretical basis for his metrics needs considerable attention. Halstead, more than other researchers, tried to integrate theory from both computer science and psychology. Unfortunately, some of the psychological assumptions underlying his work are difficult to justify for the phenomena to which he applied them. In general, computer scientists would do well to immediately purge from their memories:

- The magic number 7 ± 2
- The Stroud number of 18 mental discriminations per second.

These numbers describe cognitive processes related to the perception or retention of simple stimuli, rather than the complex information processing tasks involved in programming. Broadbent (1975) argues that for complicated tasks (such as understanding a program) the magic number is substantially less than seven. These numbers have been incorrectly applied in too many explanations and are too frequently cited by people who have never read the original articles (Miller, 1956; Stroud, 1967). Regardless of the validity of his assumptions, Halstead was a pioneer in attempting to develop interdisciplinary theory, and his efforts have provided considerable grist for further investigation.

Interconnectedness

Since the modularization of software has become an increasingly important concept in software engineering (Parnas, 1972), several metrics have been developed to assess the complexity of the interconnectedness among the parts comprising a software system (Belady & Lehman, 1976; McClure, 1978; Myers, 1975, 1978; Yau & Collofellow, 1978). For instance, Myers (1975) models system complexity by developing a dependency matrix among pairs of modules based on whether there is an interface between them. Although his measure does not appear to have received much empirical attention, it does present two important considerations for modeling complexity at the system level (Myers, 1978). The first consideration is the strength of a module; the nature of the relationships among the elements within a module. The stronger, more tightly bound a module, the more singular the purpose served by the processes performed within it. The

second consideration is the coupling between modules; the relationship created between modules by the nature of the data and control that is passed between them.

A primary principle of modular design is to achieve as much independence among modules as possible. This independence helps to localize the impact of errors or modifications to within one or a few modules. Thus, the complexity of the interface between modules may prove to be an excellent predictor of the difficulty experienced in developing and maintaining large systems. Myers' measure identifies data flow as a critical factor in maintainability. Nevertheless, his measure has not been completely operationally defined, and its current value is primarily heuristic. Yau and his associates (1979) are currently working on validating a model of this generic type. Unfortunately, little empirical evidence is available to assess the predictive validity of such metrics.

The focus of metrics measuring the interconnectedness among parts of a system is quite different from those which measure elementary program constructs or control flow. Metrics measuring the latter phenomena take a micro-view of the program, while interconnectedness metrics speak to a macro-level. An improved understanding of aggregating from the micro- to the macro-level needs to be achieved. For instance, summing the Halstead measures across modules leads to very different results than computing them once over the entire program (Milliman & Curtis, 1980).

Interconnectedness metrics may prove more appropriate parameters for macro-level models such as those which predict maintenance costs and resources. Macro-level metrics may prove better because factors to which micro-level metrics are more sensitive, such as individual differences among programmers, are balanced out at the macro- or project level. Macro-level metrics are less perturbed by these factors, increasing their benefit to an overall understanding of system complexity and its impact on system costs and performance.

Conclusions

A major theme behind the effort to develop valid measures of software quality and complexity is the desire to provide predictors of the life cycle costs and resources which must be invested in a software system. A traditional measure of life cycle effort has been lines of code developed or modified per person-month. However, this is a terribly flawed measure of productivity (Jones, 1978) which does not provide insight into the processes which affect costs and resources. Software characteristics are believed to be much more directly related to processes which determine cost and resource outcomes. That is, the research reviewed here demonstrated that several software complexity metrics are related to the number of errors left in a section of code and the ease of debugging and modifying the code. Software errors and the effort required to find and eliminate them are a major driver of software costs and resources. For instance, an error caught during system specification may be 50 times cheaper to repair than if it were not detected until system testing (Jones, 1978). To the extent that software metrics can provide feedback on error-proneness throughout the life cycle, they become a means of estimating and controlling costs and resources.

Research needs to be conducted at two levels of analysis. At the macro-level, appropriate metrics need to be developed which relate to system

costs and effort. Most existing measures are computed directly from the code. If measures can be developed which use information available during the initial requirements allocation or software system specification, they could be used in costing and sizing models. Metrics computed on tested code could be used as input to software reliability models which require an estimate of delivered errors. Metrics computed at the macro-level will be valuable for management information and prediction, but not necessarily for developing process models of software development and maintenance activities.

Software metrics computed at the micro-level, such as those of Halstead and McCabe, will probably be of greater assistance in developing models of software development and maintenance processes. Continued research needs to determine the most important characteristics affecting software development and maintenance tasks. It is also possible that the importance of these characteristics will vary by the nature of the task and the capability of the programmer. For instance, should different software characteristics prove to be better predictors of performance based on the capability of the programmer, they may provide insight into how a programmer's method of understanding software matures. Such an understanding could lead to designing better training techniques.

The current state-of-the-art is that interesting relationships have been demonstrated. Future research needs to refine the metrics, develop predictive equations, validate them on large data sets, and refine the metrics as software technology changes.

ACKNOWLEDGEMENTS

I would like to thank Laszlo Belady and Sylvia Sheppard, and Drs. Elizabeth Kruesi and John O'Hare for their thoughts and comments. Some of this paper was drawn from work supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N00014-79-C-0595). However, the opinions expressed in this paper are not necessarily those of the Department of the Navy. Reprints can be obtained from Dr. Bill Curtis; General Electric Company, Suite 200; 1755 Jefferson Davis Highway; Arlington, VA 22202.

REFERENCES

- Baker, A.L. & Zweben, S.H. A comparison of measures of control flow complexity. In Proceedings of COMPSAC '79. New York: IEEE, 1979, 695-701.
- Basili, V.R. & Reiter, R.W. Evaluating automatable measures of software development. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 107-116.
- Belady, L.A. Software complexity. In Software Phenomenology. Atlanta: AIRMICS, 1977, 371-383.
- Belady, L.A. Complexity of programming: A brief summary. In Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost. New York: IEEE, 1980, 90-94.
- Belady, L.A. & Lehman, M.M. A model of large program development. IBM Systems Journal, 1976, 15(3), 225-252.
- Bell, D.E. & Sullivan, J.E. Further Investigations into the Complexity of Software (Tech. Rep. MTR-2874). Bedford, MA: MITRE, 1974.
- Benyon-Tinker, G. Complexity measures in an evolving large system. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 117-127.
- Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., Merrit, M.J. Characteristics of Software Quality. Amsterdam: North Holland, 1978.
- Broadbent, D.E. The magic number seven after fifteen years. In A. Kennedy & A. Wilkes (Eds.). Studies in Long Term Memory. New York: Wiley, 1975, 3-18.
- Chen, E.T. Program complexity and programmer productivity. IEEE Transactions on Software Engineering, 1978, 3, 187-194.
- Cornell, L.M. & Halstead, M.H. Predicting the Number of Bugs Expected in a Program Module (Tech. Rep. CSD-TR-205). West Lafayette, IN: Purdue University, Computer Science Department, 1976.
- Curtis, B. In search of software complexity. In Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 95-106.
- Curtis, B., Sheppard, S.B., & Millman, P. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979, 356-360.

- Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T.
Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 96-104.
- Elshoff, J.L. Measuring commercial PL/I programs using Halstead's criteria. SIGPLAN Notices, 1976, 11, 38-46.
- Elshoff, J.L. A review of software measurement studies at General Motors Research Laboratories. In Proceedings of the Second Software Life Cycle Management Workshop. New York: IEEE, 1978, 166-171.
- Feuer, A.R. & Fowlkes, E.G. Some results from an empirical study of computer software. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979, 351-355.
- Fitzsimmons, A.B. Relating the presence of software errors to the theory of software science. In A.E. Wasserman and R.H. Sprague (Eds.), Proceedings of the Eleventh Hawaii International Conference on Systems Sciences. Western Periodicals, 1978, 40-46.
- Fitzsimmons, A.B. & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.
- Freburger, K. & Basili, V.R. The Software Engineering Laboratory: Relationship Equations (Tech. Rep. TR-764). College Park, MD: University of Maryland, Computer Science Department, 1979.
- Freiman, F.R. & Park, R.E. PRICE software model - Version 3 an overview. In Workshop on Quantitative Software Models for Reliability, Complexity and Cost. New York: IEEE, 1980, 32-41.
- Funami, Y. & Halstead, M.H. A software physics analysis of Akiyama's debugging data. In Proceedings of the MRI 24th International Symposium: Software Engineering. New York: Polytechnic Press, 1976, 133-138.
- Gilb, T. Software Metrics. Cambridge, MA: Winthrop, 1977.
- Green, T.F., Schneidewind, N.F., Howard, G.T., & Pariseau, R. Program structures, complexity and error characteristics. In Proceedings of the Symposium on Computer Software Engineering. New York: Polytechnic Press, 1976, 139-154.
- Gordon, R.D. & Halstead, M.H. An experiment comparing Fortran programming times with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.
- Halstead, M.H. Natural laws controlling algorithm structure. SIGPLAN Notices, 1972, 7(2), 19-26.

- Halstead, M.H. An experimental determination of the "purity" of a trivial algorithm. ACM SIGME Performance Evaluation Review, 1973, 2(1), 10-15.
- Halstead, M.H. Elements of Software Science. New York: Elsevier North-Holland, 1977.
- Jones, T.C. Measuring programming quality and productivity. IBM Systems Journal, 1978, 17(1), 39-63.
- Lehman, M.M. Programs, programming and the software life cycle. IEEE Proceedings, 1980, 68, xxx-xxx.
- Love, L.T. & Bowman, A. An independent test of the theory of software physics. SIGPLAN Notices, 1976, 11, 42-49.
- McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.
- McCall, J.A., Richards, P.K., & Walters, G.F. Factors in Software Quality (Tech. Rep. 77CIS02). Sunnyvale, CA: General Electric, Command and Information Systems, 1977.
- McClure, C.L. Reducing COBOL Complexity through Structured Programming. New York: Van Nostrand Reinhold, 1978.
- Miller, G.A. The magic number seven, plus or minus two. Psychological Review, 1956, 63, 81-97.
- Milliman, P. & Curtis, B. An evaluation of modern programming practices in an aerospace environment. In Proceedings of the Third Digital Avionics Systems Conference. New York: IEEE, 1979.
- Milliman, P. & Curtis, B. A Matched Project Evaluation of Modern Programming Practices (RADC-TR-80-6, 2 vols.). Griffiss AFB, NY: Rome Air Development Center, 1980.
- Mohanty, S.N. Models and measurements for quality assessment of software. ACM Computing Surveys, 1979, 11, 251-275.
- Morrison, D.F. Multivariate Statistical Methods. New York: McGraw-Hill, 1967.
- Myers, G.J. Software Reliability. New York: Wiley, 1976.
- Myers, G.J. An extension to the cyclomatic measure of program complexity. SIGPLAN Notices, 1977, 12(10), 61-64.
- Myers, G.J. Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
- Ottenstein, L.M. Quantitative estimates of debugging requirements. IEEE Transactions on Software Engineering, 1979, 5, 504-514.
- Parnas, D.L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, 15, 1053-1058.

- Putnam, L.H. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering, 1978, 4, 345-361.
- Rabin, M.O. Complexity of computations. Communications of the ACM, 1977, 20, 625-633.
- Richards, P. & Chang, P. Localization of Variables: A Measure of Complexity (Tech. Rep. 75CIS01). Sunnyvale, CA: General Electric, Command and Information Systems, 1975.
- Schneidewind, N. & Hoffman, H.M. An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering, 1979, 5, 276-286.
- Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Human factors experiments on modern coding practices. Computer, 1979, 12, 41-49.
- Sheppard, S.B., Milliman, P. & Curtis, B. Experimental Evaluation of On-line Program Construction (Tech. Rep. TR-79-388100-6). Arlington, VA: General Electric, Information Systems Programs, 1979.
- Stroud, J.M. The fine structure of psychological time. New York Academy of Sciences Annals, 1967, 138(2), 623-631.
- Walston, C.E. & Felix, C.P. A method of programming measurement and estimation, IBM Systems Journal, 1977, 16, 54-73.
- Walters, G.F. Applications of metrics to a Software Quality Management (QM) program. In J.D. Cooper & M.J. Fisher (Eds.), Software Quality Management. New York: Petrocelli, 1979, 143-157.
- Wolverton, W.R. The cost of developing large scale software. IEEE Transactions on Computers, 1974, 23, 615-634.
- Woodward, M.R., Hennell, M.A., & Hedley, D. A measure of control flow complexity in program text. IEEE Transactions on Software Engineering, 1979, 5, 45-50.
- Yau, S.S. & Collofellow, J.S. Some stability measures for software maintenance. In Proceedings of COMPSAC '79. New York: IEEE, 1979, 674-679.

Complexity of Large Systems

L. A. Belady

Domain of Study

At first approximation a program, including a large software system, is a piece of text, suitable for machine processing. This processing is usually compilation and results in a bit-string of zeros and ones - the object code - which, when loaded into the machine, is capable of guiding further processing of information. Obviously, the compiler itself is a program whose bit-string, derived from its text, guides the machine to process other programs' text into object code.

A program first of all must be correct. But we also require that, during execution, machine resources be economically used. Programs may be equivalent in the sense that the same input data always processed into the same output data, yet the programs may significantly differ in their execution time and memory demand on the same machine. Researchers active in the discipline of computational complexity study algorithms from this resource demand point of view. On the more empirical side, operating system designers and modelers study the interaction of programs sharing a single computer. Their purpose is to find better coordination of program execution, to reduce idleness of resources and to increase system throughput.

These studies thus consider the complex execution dynamics of programs and seek answers to questions like: which algorithm, expressed in the program text, is best for a given

machine configuration; or which resource allocation strategy, or parameter setting, is to be applied in an operating system. Notice, however, that the structure, format or appearance of the program text, and indeed the entire history of its development, the way it was created, are considered irrelevant in this context of machine performance.

At this point we would like to differentiate between simple programs and large scale software systems, based on whether the appearance and the process of creation of their documentation are considered significantly influencing the total cost of their processing data. The text of a small program is usually the intellectual product of a single person or of an informally cooperating and communicating team. In the following we will call software those systems whose dynamics of evolution demands the planned and coordinated activity of a human organization, conducted along many phases of a project, stretching over a several year long period of time.

This program evolution can be viewed as spanned by a sequence of forms, or documents, the last one being the text ready for compilation. Forms in earlier phases are, for example, the requirement documentation, specification sheets, design documents of increasing detail, and all of the above in different versions, as modifications are performed during the lifetime of the software system. The process of creating one form out of another is a human, and today mostly manual, activity. This is the reason why this process is also the major source of cost and error. Since man to man communication

is a significant component of the process, the structure, arrangement, availability, style and clarity of representation of each of these forms strongly influence cost and quality. Simply put, the program is not only the input to the computer, but even more importantly the major means of human communication during development.

The current question is: how can we capture, possibly numerically, properties of text and other software documents, to aid prediction of development cost and schedule, and the reliability of the final product, or to evaluate the impact on the above of new tools and techniques? Informally, we tend to use the word complexity to describe generically the difficulty we encounter in developing software, or the software's resistance to modifications. Note, however, that we want measures of complexity related to a human process, and not describe a machine procedure as is the case with computational complexity, or with operating system allocation studies. The latter are taken care of quite well somewhere else. In the following we discuss quantification efforts so far proposed for the programming process.

State of the art

There seems to be a consensus that complexity of programming has to do with properties beyond and other than the sheer size of program text or document. Intuitively, complexity has many other factors: preparedness of the crew (training and experience), connectedness of components, easy reach of relevant information (availability of files, libraries, experts); also, variety vs. repetition, surprise vs. predictability, - to give just a few

Ideas. Nevertheless, all these factors manifest themselves in the amount of time spent on the task. Thus time itself could be the global measure of at least the symptoms of complexity.

It is the decomposition of aggregate time, and the association of the resulting time components with particular software attributes, which are the main issues here. For example, one could treat separately the complexity of the distinct activities of specification, design, maintenance, etc. This makes even more sense if we consider that the early phases of modularly structured large software development are characterized by intermodule allocation of functions and the firming up of interfaces. In contrast, later implementation phases are involved with the intermodule aspects of text, expressed in the syntax of some language. In fact, many of the complexity metrics proposed so far and mentioned below can be for example classified into these two categories.

In general, it is more difficult to capture the complexity of creation of software than that of modification. In fact, one of the earliest published complexity metrics was deduced from observations made on the evolution - maintenance and enhancement - of modularly structured operating systems. This metric was defined to be the extent to which a change penetrated into the software, measured as the fraction of modules changed while transforming the old into a new system version. The underlying idea is that the lesser number of components are involved, the less complex, i.e. less time consuming it takes to perform modification. Probably the effect is not linear: twice as many components to work with may take more

than twice as much time.

Given the outline of the variety of activities along the life cycle phases, it is not surprising to find so many, more or relevant, papers on complexity in the literature. A recent article [] published in the Proceedings of the Workshop on "Quantitative Software Models" late last year presents a two-dimensional classification of papers related to complexity (Table 1).

Some of the papers study specifically software, and they are grouped in the upper half of the matrix. Paper in the lower half are listed because they also contain ideas applicable to software, or are quite general and thus include software systems as well.

The approaches of the papers vary significantly. Some of them are descriptive and informal, others are of quantitative nature. The latter can be further classified as being based on counting objects, expressing events in probabilistic terms, or making of organized observations. The four vertical subdivisions of the table correspond to these classes of approaches.

Another survey of the literature on complexity is an RADC report []. Although classification in this latter is somewhat different, the message coming through from the entire material is clear: we are not short of ideas but we have very few tested ideas, or validated metrics which we could safely recommend to measure and compare complexity of software. The current state is rather that of early exploration of potential approaches.

Before presenting our recommendations for short term research in complexity metrics we would like to further illuminate the state of the art through a few representative examples. To do this we quote a passage from the mentioned survey []. Other examples, or some examples amplified, can also be found in the paper on "Programming Complexity" by Curtis in this same volume.

Some Examples*

"In the category of deterministic control flow complexity the best known work is McCabe's. With the control flow graph of the program given, he proposes as complexity measure the number of distinct execution sequences which are possible along the directed graph. The application of this metric has become quite widespread, because the number of paths is easy to extract automatically from existing (and machine stored) programs. This approach also appeals to intuition: a person reading a program must mentally follow all control paths in order to fully understand the program. Unfortunately the even more complex activity of following data reference paths is completely neglected in this model."

"Another control flow based measure was proposed by Woodward, et al. The basis of their approach is program text, amended by lines which interconnect statements where control may be passed between them. These lines occasionally cross each other and thus create "knots". The complexity then is

* The examples are all referenced in the bibliography of [].

assumed to be proportional to the knot-count. Indeed, well-structured, easy-to-ready programs have less knots, but again data references are not included here (although the knot method could include data graphs as well)."

"Also related to the above approaches are Cobb's "reachability" measure, Myers' extension to McCabe's model, and an early paper by Farr and Zagorsky proposing the density of IF statement as a measure of complexity."

"Significantly different is the approach taken by Yin and Winchester. Here data flow complexity is considered basic, and the associated graph's departure from its spanning tree is defined to be the measure of complexity. The rationale is that in a tree a unique path leads to each node - a sort of minimum complexity.

"The most comprehensive of the deterministic approaches based on program object counts is unquestionably Halstead's software science. It is based on the four counts of: distinct operators, operands, and total occurrences of operators and operands in the program. From these numbers bounds and estimates of program size, programming effort, etc., are derived. The approach has received considerable attention. Often, experimentalists summarize their own results in terms of Halstead measures, or test and verify the claims of software science. Such work has been reported for example by Curtis."

"There are too many other deterministic proposals to mention them all. Due to its originality, however, we cite here Mills' proposal to measure complexity of a program by the length of text necessary to prove its correctness. (Will this motivate for simplicity?)"

"For modular systems an example is the Belady/Lehman model in which complexity of system modification is captured as the ratio of modified to total number of system components (modules). Clearly, if modifications get diffused into a larger portion of the system, then it must have been more intertwined and complex than a system in which modifications remain confined and localized. The approach has been successfully used to predict modification workload of a large operating system which has been evolving over a ten year period."

"However interesting and promising, information theory based approaches are rather rare in the literature and appear concentrated either around the study of probabilistic algorithms or of interconnected systems in general. More specifically for program systems, Belady and Bellner attempted to capture the complexity of program evolution by introducing distributions over the set of modules of the probability that (a) a change hits a given module and (b) that another module becomes impacted by the change. The scheme is formally quite close to the entropy approach of van Emden's... Another, earlier, effort by Haney models the change propagation by modules as nodes. Unfortunately, very little experience exists with these approaches and at present they are subjects of research".

"There are numerous papers in the literature on readability, complexity of comprehension, frequency distributions of words and symbols in natural languages as compared to program symbols, many based on probability theory. But again, experience with respect to their usability is practically nil."

"Much more promising are the empirical approaches. In one of the earliest studies, L. Weissman at the University of Toronto identified a number of program constructs and attributes and ranked them according to the associated relative difficulty which a group of students encountered while programming. By implication, a construct is more complex the more difficult it is to apply and understand it in a program. Encouraged by earlier results, recently more professional psychologists turned their attention to the empirical evaluation of programming complexity. They usually conduct joint efforts with computer scientists, with the objective to understand human factors of programming tools and techniques and to test in practice the measures arrived at by speculation, which would otherwise be doomed to oblivion."

Potential Impact

Although reasons for using complexity metrics are manifold, ideally one would like to extract from measurements and observations a small set of numbers which could be used to predict quality, which in turn may help estimate maintenance load, and cost of further development and enhancement. In addition, it would be important to compare and mutually learn from, projects on different products and under different circumstances -

somehow along the line of "model rules" as applied in experimental physics. Also, complexity metrics could be instrumental in deciding whether a piece of software is ready for retirement if it is so complex that modifying it would cost more than redesigning it from scratch. Note that this implies the need for know-how to predict costs of both modification and new design, the latter requiring the application of advanced tools and techniques, usually not available for work on the old version.

There are some other, perhaps less important, applications of these metrics. Programming styles could for example be classified, and programmers recognized based on programs they wrote. Similarly, language classification could become more meaningful, at least as applied to the ranking of programmer/language pairs or program/programmer/language triplets. But detailed discussion of these issues is certainly beyond the scope of this writing.

Recommended Short-term Research

As already indicated, most of the early work on software complexity involved ad hoc, isolated and uncoordinated efforts. As a result, the definition of objects (for example line of code, language constructs) in the domain of individual studies seems quite arbitrary, thus preventing systematic comparison of either different metrics, or measured values of the same metric applied to different systems or environments. Nevertheless, we should not ignore some promising results already reported, or the many efforts in progress. It is therefore necessary to

A. Standardize terms and dimensions of observable and measurable attributes, that of programs and of the development/maintenance process, such that ongoing efforts be not disrupted and, instead, exchange of experience among them encouraged. This research effort, based on already existing reports and data, should be sponsored by DoD rather than a smaller organization, in order to enlarge the data base and to enforce commonality. We already mentioned that quantification of from-scratch design and development of large software could be too hard a nut to crack. It is therefore recommended that

B. Study of patterns in trends and measurements of evolving large systems be conducted, using definitions as suggested in A. The resulting data should then be used to build models of software evolution, which in turn can be used to explore alternative approaches more rapidly and less expensively than with full scale experimentation. A by-product of this effort could be metrics for complexity of maintenance and enhancement.

We feel that probabilistic, or information theoretical, models of structural complexity, some of them studied already, should be refined by involving information theorists. Therefore we recommend that

C. Interdisciplinary research be started in the area

of structural complexity, employing computer scientists, statisticians and information theorists. Also, concepts such as locality (of information in this context), so successfully applied in program execution dynamics, should be examined and experimented with to study difficulties and the impact of tools and techniques on development and maintenance.

A final note: complexity studies, such as complexity of computation, related to the execution time behavior of programs, are in good hands, and the corresponding efforts do not need reinforcement. It is the evolution dynamics of large software which needs attention.

Bibliography

DACS Quantitative Software Models March 1979.

Belady, L. A. Survey of software complexity measures. Proceedings of the IEEE NY Poly Workshop on Quantitative Software Models, Kiamesha Lake, NY, October 1979.

Lehman, M. M. Programs, programming and the software life cycle. Special issue of IEEE Proceedings, September 1980.

Curtis, B. Measurement and experimentation in software engineering. Special issue of IEEE Proceedings, September 1980.

SOFTWARE MAINTENANCE TOOLS AND STATISTICS
IN THE LIFE CYCLE OF A COMPUTING APPLICATION*

by

MERVIN E. MULLER

Director, Computing Activities Department

World Bank**

Washington, D.C.

June 3, 1980
(Revision of May 8, 1980)

*Prepared for "Meeting on Software Metrics," Professor Alan J. Perlis, Chairman of Software Metrics Panel, National Academy of Sciences Auditorium, June 30, 1980.

**Comments made here do not represent official views of the World Bank.

Software Maintenance Tools and Statistics
in the Life Cycle of a Computing Application

by

Mervin E. Muller

Abstract

Many of the activities involved with the maintenance of computing applications suggest areas of research into needed software metrics. Improvements in software metrics, in turn, can lead to more effective maintenance. Maintenance of a computing application tends to include more activities than are appropriate, and maintenance is typically more complex and costly than desired. In most respects we lack the tools and statistical techniques to gain an adequate picture of what is going on. We need a conceptual framework for maintenance and a related set of software metrics. These objectives raise a series of research questions, some of which are addressed in this paper.

The use of a life cycle model is reviewed as an aid in understanding what software maintenance tools and statistics can offer. Appropriate perspective is sought through better understanding of the possible relationships among maintenance, enhancement, and replacement. Activities affecting the maintenance of computing applications require meaningful data and analysis, and for this purpose several concepts are introduced, namely: a) Attributed Cause and Attributed Need, b) Contamination Effects, c) Cascading-Causal Chains and Linkages, and d) Indirect and Direct Tools. The role of statistical theory and application is explored in order to indicate how statistics can help guide future research on needed software maintenance tools. Areas of needed research are outlined, together with some pitfalls to be avoided in pursuing the research challenges.

June 3, 1980, replaced version of May 8, 1980

1. INTRODUCTION

The expenditure of resources on the maintenance of a computing application can be substantial. To what extent are such expenditures necessary? Can the expenditures be reduced by different ways of carrying out the steps leading to the design, implementation, and use of a computing application? Maintenance of a computing application is a little understood subject, and therefore its true role should receive special attention. It is not particularly difficult to define maintenance as is done below in Section 2, or to specify the activities associated with it. However, it is often surprisingly difficult to collect uncontaminated data to measure the effects of various policies or factors which influence the need to perform maintenance, or to determine the amount of effort that is spent to perform maintenance. I believe that considerable research effort needs to be addressed to overcoming the obstacles mentioned in this paper if appropriate software metrics are to be developed to aid in understanding and managing maintenance activities. In particular, there is a need for software tools to aid both the maintenance activities and the collection of relevant data to support meaningful analysis. Statistical theory can be an effective aid in deciding how to collect, analyze, and present data underlying the software metrics.

2. WHAT IS MAINTENANCE?

Depending upon who is asking this question, or who is answering, one can either fail to appreciate its complexity or be so absorbed with the

complexity that useful communication does not take place. I have found it useful to answer the question by providing two definitions, maintenance status and maintenance work [Muller (6)].

Maintenance status. For practical purposes here, let us consider the set of materials, including the program (or package, or system) and all related documentation and test data, as a computing application. The application should be defined as being in maintenance status if it has been tested and distributed to the intended users on the assumption that it can provide the capabilities that were specified for the application.

Maintenance work. Maintenance work on a computing application in maintenance status is that effort spent on changing the actual programming (without enhancing the scope of the application), performing tests related to programming or programming changes, changing the documentation or test data, or providing assistance to those who have difficulty using the computing application. Maintenance work can also become necessary because of environmental changes such as the need to change controls due to equipment changes or operating system changes. In addition, maintenance work can relate to changing procedures to use the application, training people in how to use the application following a change, or to ensuring that changes have been distributed to the user and correctly incorporated into the user's installation.

As we shall see below, other types of activities can masquerade as maintenance and many distortions arise from this source.

3. Why Consider a Life Cycle Approach? The Relationships among Maintenance, Enhancement, and Replacement.

A life cycle model can help one to better understand and measure the development and maintenance of a computing application. There is a rich literature available on life cycle models, and the general conceptual framework helps to identify and relate the various stages throughout the life cycle of a computing application, illuminating the relationships among them--in particular, maintenance, enhancement, and replacement of the application. Maintenance, enhancement, and replacement effort can be given consistent definitions within a life cycle framework, although in the real world these stages in the life cycle model are often not easy to distinguish in terms of the available data. Moreover, the physical life-cycle analogy is sometimes indistinct when applied to the intellectual effort represented by a computer program. It is because it is difficult to translate the desirable characteristics of a life cycle model into the real world of collecting and analyzing data related to maintenance activities that many interesting and challenging research problems in software metrics arise, some of which are considered in later sections.

4. Factors Influencing Maintenance and the Selection of Metrics

There are both technical and psychological factors that affect what is accomplished as maintenance activity and these factors influence the possible choice of data to establish useful metrics. These factors are elucidated in the next few sections from several different viewpoints to

illustrate how data are subject to contamination effects and activities in the life cycle that precede or follow maintenance activities.

5. Attributed Cause and Attributed Need

5.1 Introduction. The title of this section identifies one type of information contamination in relation to maintenance that occurs due to psychological factors rather than technical considerations. I know of no single and consistent process to eliminate the finger-pointing that arises between the user and developer of a computing application when a user asserts that the computing application is "working incorrectly." Often one sees work on a computing application that is done as a maintenance task when in fact the resources are really being expended for an enhancement. The misclassification of effort as maintenance can arise either because the person incompletely or incorrectly specified his need in the first place, or because the developer of the computing capability has misunderstood or failed to provide the "needed" capability, or because needs have changed. What an uninvolved observer can detect, usually after the fact, is a defective communication between the "user" and "developer" of a capability about an attributed need or the attributed cause of dissatisfaction.

5.2 "Sleeping dogs" and "open patients". It is often the case that renewed user attention, or new users, will awaken the need ("sleeping dogs") for doing something about an application that has escaped attention

or has not been considered important enough to warrant formal maintenance in the past. Similarly, programmers frequently notice, during maintenance, things that need correction or enhancement (i.e., while the patient is open) and proceed to make the necessary changes. Far from being harmful, these behavior patterns may bring about significant improvements; however, they tend to distort our data on maintenance work.

5.3 Are some "bugs" acceptable? Most organizations will permit a needed maintenance change to a program, but many organizations require justification to establish why an enhancement is necessary. This adds to the information contamination. In an ideal situation, there would be no blame or no penalty to the user or developer so that one could easily separate the need to perform a maintenance or enhancement activity. Clear and complete design specifications may produce better accuracy. I believe that part of the problem arises because of the mistaken belief that a computing capability can and should be designed, implemented, and used, in a single step. For the foreseeable future I believe a partial solution to this problem would be to have computing applications evolve through a series of design-implementation-use iterations. However, this iterative process requires careful data identification to separate maintenance, enhancement, and replacement activities.

The research questions that arise here relate to setting-up reproducible human factor experiments to understand either how to avoid the "attributed need, attributed cause" syndrome and its resulting data con-

tamination, or how to adjust the data to offset the effect of human factors. One should question how much priority should be given to such research effort, because of the uncertain pay-off. However, to make significant progress in software metrics for maintenance work, this problem must be overcome.

6. Contamination Effects

6.1 Introduction. Since contamination of data may result when users and developers of a computing application treat an enhancement requirement as a maintenance requirement, this problem might be reduced or avoided by having the maintenance done by someone other than developers, though this creates an inevitable transfer cost. However, there are also other forms of misclassification. Various stages in the life cycle affect tasks defined as maintenance, and therefore contribute to the contamination of data identifying work done as maintenance. There are other types of contamination that also need to be understood and identified before research on software metrics can aid designers and maintainers of computing applications. Some are discussed below.

6.2 Cumulative Effects of Changes. The life cycle model seems most appropriate for a mechanical device subject to wear. For a computing application, the analogue of wear is the process of "deterioration" that occurs to a computing application as a result of maintenance or enhancements, or accommodation to shifting user needs. The deterioration or

quasi-deterioration can be in the programs, test data, or documentation. The end result is that a program or piece of documentation no longer has an evident integrity of design, or perhaps that the coherency of its design is no longer present. It could be argued that "pure" maintenance activities should not affect the useful life of a computing application if they were not contaminated by a concealed element of enhancement. Unfortunately, several myths have developed around typical attempts to maintain programs. There is a prevalent belief that computing applications become unmaintainable because of the cumulative effects of maintenance activities. Many applications are therefore replaced because the "old programs can no longer be maintained." With the contamination of data related to maintenance, enhancement, and replacement, we find little valid data available. Maintenance must be studied in relation to the place it properly occupies in the life cycle. What is needed is research to measure and elucidate what is meant by program "deterioration." The purpose of this research should be to provide guidance on how to perform maintenance rigorously without disturbing the integrity of the program. The research itself can become contaminated, if it fails to take into account the maintenance of test data and documentation consistent with the program.

A life cycle model with sufficient detail might also help to distinguish between "deterioration" and "obsolescence" with respect to maintenance. Even if programs, test data, documentation, or controls, have not deteriorated, their future maintenance cost could be significantly reduced if they were replaced by newer techniques--for example, higher level

languages, table or parameter driven programs, data base systems, test data systems, a documentation system, or newer job control systems. In order to justify the cost of making replacements, one needs to forecast future maintenance demands, and this requires valid data and models.

The life cycle model can facilitate making reliable predictions of the cost of various stages of a computing application. Such cost estimates are useful in deciding to replace rather than maintain an application--i.e., if the cumulative actual cost of maintenance is near or beyond that amount adopted as the criterion for eliminating or replacing the application.

6.3 Distortions due to faulty use of controls or documentation.

When measuring the effort spent on maintenance activities, the data must be identified with sufficient precision to distinguish effort spent in attempting to detect alleged bugs that are thought to exist because the user failed to understand the controls or the documentation. The resolution of such misunderstanding can require significant amounts of time. Such effort should be a signal to review the controls and documentation to determine if they are faulty or themselves need maintenance. Time spent on such evaluations and time spent on modifying controls, including preparation of clearer error messages, or documentation, should itself be considered a necessary maintenance activity but one of a different kind. Software metrics should permit separate identification of the time spent on detecting such limitations, correcting the limitations, and making pro-

gram changes if they are really required.

6.4 Distribution Effects. When a computing application is distributed to multiple users, usually at multiple locations and possibly using different operating systems, one should be aware that possible distortions can easily occur in the data related to maintenance. There are many opportunities for contamination. For example, some users may fail to have releases of the controls, programs, documentation, or test data leading to distortions such as those mentioned above in Section 6.3. More subtle problems occur when a sequence of several maintenance items have been distributed and not all have been incorporated, or some have been incorporated incorrectly. These problems can be attributed to defective administrative procedures of distribution; however, there may also be a technical failing: if the computing application had contained sufficient diagnostic capabilities, the omissions or errors might have been detected, perhaps should have been. Therefore, one research problem ought to be how to specify effective diagnostic capabilities. The challenge of software metrics in this situation would be to measure the cost-effectiveness and necessary trade-offs in development of needed diagnostic capabilities.

6.5 Diagnostic Limitations of Test Cases and Data. How can one measure the cost of incorporating diagnostic capabilities in an application, including both the design and maintenance of such capabilities, and the overhead burden of executing the application when the diagnostic capa-

bilities are included? Without better awareness of the cost of the problems cited in earlier sections, one will have difficulty in estimating the benefits derived from the presence of adequate diagnostic capabilities. A related activity of building and using these diagnostic capabilities is the effort required to modify them as the maintenance of a system requires changes or incorporation of additional diagnostic capabilities. An additional contamination effect may be introduced because changes to the diagnostic capabilities may not be truly maintenance activities but in fact enhancement or replacement activities.

6.6 Malfunctions due to Equipment or Operating System. Often the consequences attributed to a maintenance activity can subsequently be traced to a problem with the equipment or the software of the operating system. Adequate diagnostic tools, such as those mentioned in Section 6.5, can aid in identifying and separating the causes of the problem. It is very tempting for those involved with maintenance activities to classify a problem as due to the hardware or the operating system. However, the result may be to mask a deficiency in the application or to contaminate the data by reporting that time spent to fix a deficiency was spent to ascertain that the cause of the problem was the hardware or operating system. One could argue that if the computing application is well designed there should be few, if any, instances when the cause of a reproducible problem due to hardware or the operating system cannot be quickly detected. The question of the confounding of such problems with problems

with the application, and how to distinguish these problems is an important challenge for those developing software metrics.

6.7 Unintended or Unexpected Uses of the Application. Often one can observe the spending of considerable effort on maintenance activity, only to discover afterwards that an attempt was made to use the application in an unforeseen way by using the controls in an unanticipated manner or providing data which created unforeseen calculations. The situation raises additional questions of how to ensure adequate data identification. It is a demanding research problem to specify what kinds of software metrics will be useful here. Even with valid data, one still confronts the often-mentioned confusion over whether or not to make changes ostensibly as a maintenance activity to cope with the user's problem.

6.8 Errors Introduced by Maintenance. If a sequence of maintenance tasks has been performed and it is later discovered that one or more of them are in error, should further corrective maintenance effort be attributed to the design and implementation of the application or to faulty maintenance efforts? Data contamination from this source creates a serious difficulty to those who want to understand the interactions among design, implementation, maintenance, enhancement, and replacement. The need for software metrics is undisputed, but it is not clear how best to proceed. Some of the difficulties relate to the cascade effects addressed in the next section. See also Section 8.

7. Cascading-Causal Chains and Linkages

The life cycle model is a formal way of acknowledging the interdependence of the various stages of work on a computing application from the inception of the planning. However, the model by itself does not have sufficient granularity to provide adequate event identification so that an action taken in an early stage can be traced to show its impact on subsequent stages. The interdependence of events at different stages needs more attention than at present, and a much richer data structure will be needed in order to identify events and the associated data. Physical phenomena are sometimes identified and analyzed as a set of cascading events with linkages or causal chains. In such a formulation, under some reasonable assumptions, one can sometimes describe a process as a sequence of transition events, simplifying both the data collection and storage, and the data analysis. This type of formulation may be useful to understanding how design and implementation alternatives influence maintenance. Otherwise one must consider analytical approaches, which are far more complex, or lack sufficient detail to provide insight. A useful research question is whether or not such formulations are adequate. Until such research on software metrics is undertaken, we will hear many claims about the virtues of one or another particular approach, with little if any data to support or reject the the proposed approach, be it top-down, bottom-up, structured, or something else.

Until we support research on metrics we will find it difficult to justify any particular design or implementation approach. For example, I

believe that a computing application which is based on a modular design is easier to design, test, document, and maintain than one that does not use a modular design. However, there is little actual evidence bearing on this question, one way or the other. Furthermore, without diligent study we will not know why a program module included in a modular design fails to maintain its modularity during modification as part of a maintenance or enhancement effort. Without further development of software metrics, one can expect to see a continued flow of misleading or inconclusive results on this question, because of contamination effects that could have been taken into account through a cascade or causal-chain approach.

8. Indirect and Direct Tools to Aid Maintenance

Two classes of tools are needed to aid our understanding of the maintenance process and to aid in establishing software metrics. The tools are needed to support the collection of data of the comprehensive types suggested in earlier sections. Without such tools it is unrealistic to expect that the burden of data collection, retention, and analysis will take place at a level and on a scale that is needed.

Indirect tools to aid maintenance should come out of efforts to establish software metrics. These tools are indirect since they involve actions taken at other stages of the life cycle. Such tools should generate information to influence policy and planning decisions--for example, decisions to design and implement computing applications based upon using modularity of design, to require that applications include data collection

capabilities to diagnose performance and to aid maintenance, or to develop an application as a sequence of iterative steps.

Direct tools are those that immediately affect how maintenance is done. These tools relate to the collection, retention, and analysis of valid data about the maintenance process, whether it be administration, control, programming, testing, documentation, training, or user assistance. [See, for example, Allen (1)]. One of the most promising tools is the integration of the development and maintenance of programs with the development and maintenance of documentation. An example is the Programmer's Workbench, developed for use on UNIX. [See, for example, Bianchi and Wood (2), Dolotta and Mashey (3), Feldman (4), and Ivie (5)]. With such tools it should be possible and cost-effective to retain an institutional history of all changes and to support a rich enough coding structure to analyze cascading or causal-chains as described in Section 7. However, to make these tools effective, one must use them from the inception of the design of a computing application. One must adopt them out of appreciation for the interdependence of the various stages of the life cycle. Such tools must be included as an integral part of organizing how one performs maintenance activities. Otherwise, I fear that the identifying and collection of data on what takes place as maintenance activity will be viewed as an avoidable overhead cost or as an unnecessary burden on those doing maintenance, and the information will not receive adequate attention.

9. Human Factors

In earlier sections I have already mentioned how data related to maintenance can be contaminated as a result of failing to address human factors. The need to address the influence of human factors on developing effective software metrics is clear. What is not clear is how to proceed. Human factors often affect what we are trying to measure. To understand the implications of this for software metrics, we will have to establish well-designed and reproducible experiments. Such experiments present a challenge that is shared with all of the social sciences that depend upon statistical theory and techniques in order to make general inferences from a set of experiments. Much of the current literature on measuring the influence or the quality of effort devoted to software development or maintenance raises more questions than useful answers. One area where measurement seems to be effective is the evaluation of programming aptitude. [See Wolfe (7) and (8).] In this case, for example, using measured aptitude as a control variable, one could introduce a series of interesting questions concerning how programmer aptitude affects maintenance activities.

10. Role of Statistics

The possible role of statistics in software maintenance is suggested by the concern of statistics with the theory and application of methodology for efficient and effective data collection and analysis. Thus in many ways, the modern theory of the design of experiments can aid in

experimentation to find good software metrics or in understanding the influence of human factors. Another part of the theory of statistics is related to modeling. I believe that modeling, based upon statistical techniques, will help us to: a) cope with and understand the complexity of software metrics; b) develop methods by which we can predict the influence of factors affecting software; c) aid those making decisions affecting investments in software, and d) provide tools for selecting among alternative methods for developing software.

If one accepts the usefulness of the life cycle approach to model building for software metrics, then I believe that the role of statistical techniques is vital to exploring model building as an iterative process. In particular, statistical techniques can aid in each of the iterative steps of conjecture, model building, data collection and analysis, model verification or modifications based upon model experiments, and model acceptance and use.

11. Research Problems

Consistent with the objectives of the meeting for which this paper has been prepared, several research questions have been posed, following an explanation of why specific software metrics are needed and how this generates a need for research.

Let me review some of these questions:

- . Section 3 - The complex interactions among maintenance, enhancements, and replacement. The need for

better data and analysis to understand what takes place.

- . Section 5 - Attributed Cause and Attributed Need--
determine whether to have a one-step or iterative approach to developing computing applications.
- . Section 6 - Contamination Effects--cumulative effects of changes, distortions due to faulty controls or documentation, what is meant by deterioration, distribution effects, diagnostic data limitations, separating the influence of hardware and software malfunctions, effects of unforeseen uses, and error corrections introducing new errors.
- . Section 7 - Cascading - Causal Chains and Linkages--how to collect, retain, and analyze data when the events that occur as maintenance depend upon events for an earlier stage of the life cycle.
- . Section 8 - Tools - the need for research to design tools.
- . Section 9 - Human Factors--an almost unlimited horizon of interesting research questions.

In summary, I have suggested that use of a life-cycle model should aid the development of needed software metrics. However, many research questions persist. For example, how can one be sure that with a life-cycle view of development, software metrics can indicate when to replace,

extend, or correct existing software? Can software metrics aid in deciding how to replace, extend, or correct software? Merely defining the universe of discourse may be difficult--for example, how does one distinguish between an extension, correction, or replacement of software? We need to agree whether or not this distinction is important to the development of software metrics, though I believe it is.

If we can agree on the importance of these distinctions, we should next agree on the need for research to identify:

- . what factors contribute to achieving good software
 - development
 - use
 - maintenance
- . what factors contribute to accurate cost estimates of
 - development
 - use
 - maintenance
- . what factors contribute to establishing valid specifications to be met by a particular software product.

12. A Closing Remark

I hope that this paper has helped the reader to face twin questions: why are software metrics needed? How can one proceed to establish needed metrics? I was pleased to be asked by the meeting organizers to concentrate on this neglected topic and to suggest research questions. Without adequate attention to maintenance, we will continue to see unnecessary and wasteful effort spent on maintenance, as well as premature replacement of "old" applications with "new-replacement" applications without building

upon what already exists. I believe software metrics is a rich and challenging field for research and for the development of practical and useful tools.

ACKNOWLEDGMENT

I wish to thank George W. Barclay, Daniel Hoyle, and Leonard Steinberg for their helpful comments on an earlier draft of this paper.

REFERENCES*

- (1) Allen, J. R. (1977), "Some Testing and Maintenance Considerations in Package Design and Implementation," Computer Science and Statistics: Tenth Annual Symposium on the Interface, eds. David Hogben and Denis W. Fife, U.S. Department of Commerce/National Bureau of Standards, Washington, D.C., 211-214.
- (2) Bianchi, M.H., and Wood, J.L. (1976), "A User's Viewpoint of a Programmer's Workbench," Proceedings of the Second International Conference on Software Engineering, Institute of Electrical and Electronic Engineers, San Francisco, 193-199.
- (3) Dolotta, T. A., and Mashey, J.R. (1976). "An Introduction to the Programmer's Workbench," Proceedings of the Second International Conference on Software Engineering, Institute of Electrical and Electronic Engineers, San Francisco, 164-168.
- (4) Feldman, S.I. (1977), "Make - A Program for Maintaining Computer Programs," Computer Science Technical Report No. 57, Bell Laboratories, Murray Hill, New Jersey.
- (5) Ivie, E.L. (1977), "The Programmer's Workbench -- A Machine for Software Development," Communications of the Association for Computing Machinery, Vol. 20, No. 10, 746-753.
- (6) Muller, M.E. (1977), "Maintenance and Distribution of Statistical Software: Satisfying Diverse Needs," Proceedings of the Computer

* Note: Also consult general bibliography in the Proceedings prepared by Shaw.

Science and Statistics: Tenth Annual Symposium on the Interface
eds. David Hogben and Denis W. Fife, U.S. Department of Commerce/
National Bureau of Standards, Washington, D.C., 205-210.

- (7) Wolfe, J. M. (1971), "Perspectives on Testing for Programming Aptitude, Communication, Association for Computing Machinery, Proceedings 1971 Annual Conference, Chicago, Illinois, 268-277.
- (8) Wolfe, J. M. (1972), "A Validation Study--Long Range Predictive Capabilities of the Aptitude Assessment Battery: Programming Test," Programming Specialists, Inc., Brooklyn, New York, 13 pages.

A Scientific Approach to Statistical Software

Ivor Francis
Cornell University

Abstract

This paper supports the scientific approach to the study of all phases of the development and use of software, with particular reference to statistical software. It argues the case for including the user's viewpoint in describing and evaluating software. This requires that quantitative measures be defined to characterize a user's problem, to report the accuracy of computed solutions, and to assess the usefulness of the output. It requires that a set of standard test problems be assembled so that experiments can be conducted to compare the performance of software systems.

1. A Science of Software

In the early days of computer science the emphasis, as the name implies, was on the computer, the hardware and the algorithmic implications of the machine characteristics. The person who wrote a program would very likely be the person who used the program. Little attention was given to user conveniences and protection, or to readability and documentation of software. But as programs began to be exported, attention was focused more on these software features. More recently,

as programs have grown into very large software systems, interest has turned to the human, organizational, and managerial factors that influence the quality and cost of developing good software.

Thus computer science has grown from a study of hardware to include all these aspects of the emerging science of software. Marvin Denikoff, in his introduction to the first meeting of this Panel in New Haven, proposed that what is needed is the development of a science of software to provide a methodology or justification for choices between various computing software systems, either proposed or existing. Alan Perlis, in his statement of purpose to the Panel in June 1979, asked the fundamental question, "Can there be assigned to software and the processes associated with its design, development, use, maintenance, and evolution, indices of merit that can support quantitative comparisons and evaluation of software?"

The charge to this Panel, therefore, was to outline areas of study and a structure for a science of software. The list of areas suggested by this Panel will undoubtedly be added to in the future. In 1977 Halstead [1] used the term "software science" in a restricted sense. Fitzsimmons and Love [5], in reviewing his work, said that "the basic concepts of software science [are] program length, program volume, program level, language level, effort, and programming time," and are "based entirely upon measures which can be computed automatically from a computer program."

To this list we can add many other concepts, as other papers from this Panel show, including program complexity, language characteristics,

problem specification and performance evaluation, maintenance, accuracy and user convenience, micro modelling of program development resources, and organizational and managerial aspects of program development. The Panel has chosen to adopt the concept of the "life cycle" of software to organize these concepts, to be a model or paradigm for this science of software. This model, borrowed from biology, will no doubt be supplanted in time by an indigenous one, but it will serve for the immediate object of organizing the subject, to aid our comprehension and communication. For science is "a mental construct, by means of which a collection of objective data is arranged in a model and expressed linguistically for certain ends" (Dingle, [4]). The ends proposed by Denikoff and Perlis above are the quantitative comparisons and evaluation of existing or proposed software. The ultimate end is the improvement of the state of the software art.

2. Characterizing Software

Having adopted the life cycle model to organize this science of software, we can begin to characterize and describe the components of software which are, broadly speaking, 1) specification, 2) development, 3) testing, and 4) maintenance. Lists of the characteristics of each of these components can be drawn up. We can also group the people who will interact with this software into 1) managers, 2) computer scientists, 3) programmers, and 4) users.

We may then attempt to compile a list of desirable properties for these lists of characteristics. But the desirability of some property may depend on which group of people one is addressing, and what the intended application is. Furthermore, some tradeoffs will have to be made, for example between low cost and ease of use.

From these characterizations and desirable properties, possibly with the help of some mathematical models, we can propose indices of merit to support quantitative comparisons and evaluation of proposed or existing software.

One activity which is central to the comparison of software systems is the evaluation of performance, which consists of specification plus testing. The term "performance evaluation" has been used to measure completion time of workloads in certain machines, which is of paramount importance to computer scientists. But to users, "performance" means more: the user is also interested in the nature of the output from a software system in response to the practical problem facing the user. He is interested in the content and the form of the output, its accuracy and usability.

In the next section we discuss these two aspects of the software cycle from a user's point of view, and illustrate them with examples of quantitative measures, or metrics, which have been used to make comparisons and evaluations of statistical software.

3. Quantitative Measures of Performance for Statistical Software

Two questions that a potential user of some software will ask are, "Will it accurately compute solutions to my problems?", and "Is it sufficiently convenient to use, and does it present results in a sufficiently useful form, that it will be worth my investment of money and time in using it?" This user is asking for an evaluation of performance which will require testing the software in the light of the specification of his problems.

For many types of software applications the "accuracy" of a program's output is well defined, for example sorting, information retrieval, and maintaining payrolls: there is one correct solution. For some applications, however, there may not be a single correct answer, and the notion of "accuracy" has to be defined. This is the case of many applications in statistics.

An example of the need and usefulness of quantitative measures is provided by the study of multiple regression programs. Longley [12] compared the performance of several, widely-used multiple regression programs by submitting a single test problem to each program, and qualitatively comparing their computed solutions with an extremely precise solution which he obtained using special subroutines on an extended precision machine. Chambers [3], in comparing the accuracy of various algorithms for regression calculations, used the minimum number of significant figures in regression coefficients that were in agreement with those of a "best" program as a quantitative measure of accuracy of an algorithm.

Beaton, Rubin, and Barone [1], however, showed that Longley's precise solution was not the best solution for that test problem, and they defined a "perturbation index" as a quantitative measure of the difficulty of the problem to be solved, which incorporated the variance of the data and the word length of the machine. Velleman and Francis [14] argued that the use of the minimum number of correct significant figures was inadequate to characterize the accuracy of a solution and proposed other quantitative measures of accuracy.

Velleman, Seaman and Allen [15] used these measures of difficulty of problems and these measures of accuracy of solutions in an experiment which measured the accuracy of several programs as a function of the difficulty of a sequence of problems. Thus it was possible to plot the "response curve" for a program, the accuracy of its computations for problems of increasing difficulty. One would expect that accuracy would decline monotonically with increasing difficulty, but this was not always the case.

From such response curves it is possible to propose an index of merit incorporating both accuracy and difficulty to evaluate performance of competing programs, although a user would probably want to include measures of cost and convenience into an index of merit. Note that a physicist with very precise raw data may place greater weight on accuracy in his index than an economist who might prefer to use a cheaper program, provided it gave sufficient accuracy for well conditioned problems and advised the user when it encountered problems that were too difficult for it.

Quantitative measures of convenience and usefulness have also been used in comparing statistical systems. For example, Bryce and Hilton [2] conducted an experiment to compare the difficulty of installation of a number of statistical packages. They measured the effort of three systems programmers in installing each of three packages, using a Latin square experimental design. Francis and Valliant [10] describe an experiment to compare the ease of use of two packages for novices, using a simple cross-over design. Thisted [13] performed a further experiment to assess the adequacy of user documentation and control languages of three packages for an audience with some computer experience and statistical training. Again, he used a cross-over design. Finally, Francis and Sedransk [9] describe experiments, to compare the performance of software for analyzing survey data. Indices of merit were computed to measure tabulating power and the simplicity of the user language. These two concepts, which are apparently qualitative, required detailed, although admittedly debatable, definitions in order for them to be represented quantitatively.

Quantitative representation of qualitative properties also made possible a comparative review of over one hundred statistical systems, which used a model based on the "life cycle" of a statistical analysis to organize users' ratings of fifty-five program features, and compare them with the corresponding developer's ratings. (Francis [7])

All of these experiments, and others listed in the bibliography of Francis [6], illustrate the need for the careful use of the techniques of experimental design. Statisticians have a particular role to play here in the choice of test problems and in removing the effect of the many controllable and uncontrollable factors from the comparative measures of performance.

4. The Effect of Evaluation on the Quality of Statistical Software

In 1973 the Section on Statistical Computing of the American Statistical Association (ASA) declared its concern for the accuracy of statistical software. It established a committee which prepared a report summarizing the opinions of over one hundred contributors, in which desirable characteristics of statistical software were listed and discussed. (Francis, Heiberger, and Velleman [8]) Since then there has been considerable improvement in the quality of at least the major statistical program systems. During this time sessions on the evaluation of statistical software have been held at virtually every annual conference of the ASA and of the Symposium on the Interface of Computer Science and Statistics. A growing interest has been seen in similar sessions of COMPSTAT, the European Symposia on Computational Statistics.

The publication of the results of comparisons and evaluations of statistical software has an effect both on developers and on users. No standards have been established for either the development or the use of statistical software. Some developers have given insufficient attention to the accuracy of their product and to methods of protecting the user

against his misusing the program. Users, on the other hand, in publishing the results of analyses in which computers have been used, typically fail to identify precisely the software and hardware used. Publication of software comparisons and evaluations encourages the developers to improve their products, and prompts users, and more particularly editors and referees of journals, to take a closer look at which packages are being used in statistical analyses.

The goal of these comparisons and evaluations is to promote both the best use of existing statistical software and the development of better software. Through the comparisons of the performances of packages in performing standard test problems, de facto standards for statistical software, both existing and proposed, are established, standards which will evolve over time as the state of the software art improves.

5. Further work

This paper has argued the need to evaluate software performance with a user's perspective, that is, to specify and characterize the difficulty of the problem, to test the accuracy of computed solutions, and to assess the usability of the output, with particular reference to statistical software. All three components of evaluation call for quantitative measures. Implied also is the need for a set of standard test problems and solutions which can be used to evaluate existing software and write specifications for new software.

There is a need for well-designed experiments to evaluate certain features of software, particularly experiments which compare at least

two software systems. There is a need for these experiments to be replicated under different environments.

To capture the many dimensions of problem difficulty, accuracy of solutions, and usefulness of output by a few quantitative measures presents challenges both applied and theoretical. To carry out the many and continuing experiments necessary to monitor the quality of software will require the contributions of many researchers.

References

- [1] Beaton, A. E., D. B. Rubin and J. L. Barone, "The Acceptability of Regression Solutions: Another Look at Computational Accuracy," Journal of American Statistical Association, 71, 353, 158-168 [1976].
- [2] Bryce, G. R. and H. G. Hilton, "Local Installation of Packages," Proceedings of the Statistical Computing Section, American Statistical Association, 13-15, [1975].
- [3] Chambers, J. M., "Linear Regression Computations: Some Numerical Statistical Aspects," Bulletin of the International Statistical Institute, 45, Part 4, 245-254, [1973].
- [4] Dingle, H., "The Rational and Empirical Elements in Physics," Philosophy, 13, 148-65, [1938].
- [5] Fitzsimmons, A. and T. Love, "A Review and Evaluation of Software Science," Computing Surveys, 10, 3-18, [1978].
- [6] Francis, I. (editor), A Comparative Review of Statistical Software I: The International Association for Statistical Computing Exhibition of Statistical Software, IASC, Voorburg, Netherlands, 658 pages, [1979].
- [7] Francis, I., "A Taxonomy of Statistical Software," to appear in COMPSTAT 1980: Proceedings in Computational Statistics, Vienna, Physica-Verlag, [1980].
- [8] Francis, I. R. M. Heiberger, and P. F. Velleman., "Criteria and Considerations for the Evaluation of Statistical Program Packages," American Statistician, 29, 1, 52-56, [1975].

- [9] Francis, I. and J. Sedransk., "A Comparison of Software for Processing and Analyzing Surveys," Bulletin of the International Statistical Institute, 48, [1979].
- [10] Francis, I. and R. Valliant., "The Novice with a Statistical Package: Performance Without Competence," Proceedings of Computer Science and Statistics: 8th Annual Symposium on the Interface, 110-114, [1975].
- [11] Halstead, M. H., Elements of Software Science, New York: Elsevier North-Holland, Inc., [1977].
- [12] Longley, J., "An Appraisal of Least Squares Programs," Journal of the American Statistical Association, 62, 819-841, [1967].
- [13] Thisted, R. A., "User Documentation and Control Language: Evaluation and Comparison of Statistical Computer Packages," Proceedings of the Statistical Computing Section, American Statistical Association, [1977].
- [14] Velleman, P. F. and I. Francis., "Measuring Statistical Accuracy of Statistical Regression Problems," Proceedings of Computer Science and Statistics: 8th Annual Symposium on the Interface, 122-127, [1975].
- [15] Velleman, P. F., J. R. Seaman, and I. E. Allen., "Evaluating Package Regression Routines," Proceedings Statistical Computing Section, American Statistical Association, 82-83, [1977].

When is "Good" Enough? Evaluating and Selecting Software Metrics

Mary Shaw
Computer Science Department
Carnegie Mellon University
Pittsburgh, Pa.

June, 1980

Abstract: In assessing the use of metrics for software, it is important to consider the quality of the metrics themselves. This has two components. First, we can determine some of the statistical properties of the metric itself. Second, we can assess the way a metric will be used and select one that provides appropriate information without excess expense. This note discusses some issues about the validation and efficiency of measurement techniques.

1. Introduction

This note addresses two questions related to the judicious use of software metrics. The first is how proposed metrics should be described and evaluated. The second is how to select metrics that are cost-effective, in the sense that they strike a reasonable balance between the amount and precision of the information delivered and the cost of collecting and processing raw data.

The first question is of concern to people who develop metrics. It pertains to the criteria that should be used for evaluating metrics and the guidance that should be provided to prospective users. It covers not only ways to state properties of individual metrics, but also ways to validate the models that underlie the metrics and ways to decide when it is appropriate to introduce new metrics.

The second question is of concern to people who use metrics. It pertains to criteria for deciding when imprecise measures offer good enough results -- that is, to ways to determine the cost-effectiveness of a metric. Many people succumb to the temptation to demand as much detail as technologically possible, neglecting the costs of acquiring the knowledge which arise from its acquisition, comprehension, and use. We have many examples of detailed models and complex metrics; we should also look for some metrics that are both less expensive and less precise.

In a broader sense, I am discussing the problem of developing literacy and taste in the construction and use of metrics. At issue is a piece of culture -- the attitude that we as scientists and engineers take toward all the work we do, the expectations that we as readers have of technical material, and the standards that we as editors enforce for material under consideration for publication.

2. Evaluating Proposed Metrics

The most desirable metric is a direct measurement of the property of interest. Unfortunately, such measures are rarely available; they may be unavailable at the time they are needed, they may be expensive to obtain, or there may be no known way to take a direct measurement. As a result, we frequently resort to indirect measurements. When we do, we become dependent on an analytic or empirical model of the system or process being measured.

It would be nice to have a common set of program dimensions -- a "basis" -- that is commonly accepted by all investigators as the object of measurement. Unfortunately, we are far from agreement on what might constitute such a basis for the measurement of software, so we tend to develop models that are idiosyncratic in their choices of source data. In another paper in this report, I argue that a serious attempt should be made to encourage the development of a common, consistent paradigm.

Whenever a metric is indirect -- based on a model -- two issues are involved in its evaluation. The first is validation of the underlying model; the second is determination of the properties of the statistics provided by the measurement technique. In the case of a direct measurement, only the latter is of concern.

Part of the validation of a proposed metric should include a comparison of its results with the results given by existing metrics. In addition to agreeing with other ways of measuring the same thing, a new metric should clearly provide an improvement over previous techniques along at least one dimension (e.g., some property of the measure or of the cost of its use).

We must exercise discretion in the standards we set, however. Some measures are based on static analysis of program text; these will be deterministic. Other measures are based on data gathered by monitoring the execution of a program; although in a strict sense these may be deterministic, it is often not practical to treat them that way. Still other measures are based on human performance, which is much more variable and far from deterministic. Different standards must be applied in all these cases. If, for example, correlation coefficients are used to evaluate metrics, higher correlations should be expected of deterministic than of nondeterministic metrics.

2.1. Properties of a Metric

We need to establish criteria for the evaluation of proposed software metrics. This section surveys several properties that are recognized in other areas of measurement and may be pertinent for software.

When a metric can be cast as a traditional problem of statistical inference involving estimation or hypothesis testing, standard statistical techniques (e.g., [5] Chapters 9-11) can be applied. These are

sufficiently familiar that there is no need to review them here.

Some less familiar concepts from psychometrics (e.g., [8] [6]) are also worth considering. These may apply directly, particularly in productivity studies, or they may merely be suggestive of areas where we need to establish criteria of our own. I will describe criteria for *reliability* and *validity*.

A reliability measure for a metric indicates how much confidence we can have in the value of the metric for a particular individual being measured -- a program, person, system, team, or whatever. It can be thought of as a consistency measure -- an estimate of the degree to which repeated measurement of an individual would produce the same result. The reliability of a metric may be reduced by variability of several kinds:

- *Observer variability* refers to the possibility that the person taking the measurement may not always record the same events in the same way. It is unlikely to be an issue for an objective or analytic metric, but if subjective judgements are required of the observer, they may lower reliability. If the observer is a machine, this may be referred to as *instrument variability*.
- *Subject variability* refers to systematic or random changes in the individual being measured. For example, day-to-day variations in an individual's typing speed and improvement of his skill through time can both introduce variability.
- *Environmental variability* refers to uncontrolled factors that may affect measurement. It is common, for example, for the system load in a time-sharing system to affect the reliability of timings.

A number of methods are available for estimating the reliability of objective tests. The situation appears to be less well under control when the possibility for much variability exists.

A validity measure for a metric indicates how well it measures what it is supposed to measure. Several kinds of validity are recognized:

- *Content validity* or *face validity* is demonstrated by arguing that the questions asked or the data collected is pertinent to whatever is being measured. In other words, content validity can be established by getting an expert to say that the test measures what it claims to.
- *Predictive validity* is demonstrated by showing a correlation between the result of the metric and some criterion based on the future performance of the individual.
- *Construct validity* is an issue when a property is measured indirectly. Such a measurement depends on knowing a relation between the measured variables and the property of interest. Construct validity is demonstrated by showing that this relation does in fact hold.

In one sense, a validity measure should be straightforward, for validity is simply the correlation between the metric and some criterion. In practice, however, the criterion values may be expensive to

obtain or not immediately available.

It is important to note that reliability and validity are independent concepts: a test can be quite reliable without being at all valid and conversely. In other words, the results of two tests may differ because of measurement error or because different things are being measured. The two effects should not be confused.

In addition to using general criteria as a matter of course, it is appropriate to set up specialized criteria for particular classes of metrics. For example, Roberts [11] studies techniques for evaluating computer-implemented text editors. She begins by setting up standards against which to measure the techniques. In the abstract she says,

This thesis explores the possibility of performing an objective and thorough, but quick, evaluation of text editors from the viewpoints of their various users. The criterion of *objectivity* implies that the evaluation scheme not be biased in favor of any particular editor's user's model, its way of regarding text and operations on the text. As much as possible, data should be gathered by observing people who are equally familiar with each system, and when tests are analytic, the criteria for scoring must be unambiguous. *Thoroughness* implies that many aspects of editor use be considered. *Quickness* means that the tests are usable by editor designers, managers of word processing centers, or other non-psychologists who need this kind of information but have limited time and equipment resources.

In the thesis, she develops ways to answer several different categories of questions. She develops empirical methods and matching analytic (predictive) models, then shows how well the (cheaper) analytic methods do as substitutes for the (more expensive) empirical ones.

2.2. Models that Underlie Metrics

We have learned over the years that programs are easier to deal with when we abstract from details of the code to the important properties of the computation. The same principle applies to the problem of managing the complexity of the *measurement* of a program. When we introduce a model of a system, we are identifying certain properties as being of interest and suppressing the rest -- which is precisely the same approach we use for program development. Kleinrock [9] has provided fundamental queueing-theoretic models for performance evaluation. Spirn [12] has used modeling extensively for evaluating paging systems. Svobodova [13] presents a more general description of modeling techniques, but does so in a heavily hardware-oriented setting.

In an *empirical model*, equations are fit to observed data in an attempt to capture the information contained in the data. Such models can be validated by determining the correlation between the equations and observed data (fresh data or the data on which the equations were based). A variety of empirical models can be found in [2].

An *analytic model*, on the other hand, proceeds from an assumption that the process being

modelled is well understood -- at least to the extent that a mathematical description of the mechanism can be written down. Parameters of such models may be tuned to correspond to actual systems, but the underlying description may be selected as much for its mathematical tractability as for its exact correspondence to observed data. Queueing-theoretic models for operating systems [9] exemplify this approach.

Whether a model is simple or complex, empirical or analytic, it is necessary to determine how accurately it matches reality. The techniques for validating models seem to be as varied as the models. It appears that efforts to find tractable, perspicuous models that can be re-used extensively in a variety of situations have considerable payoff potential.

The importance of validating a model critically can be demonstrated by an example. Feller [4] relates a cautionary tale:

The logistic distribution function

$$F(t) = 1/(1 + e^{-\alpha t - \beta}), \quad \alpha > 0 \quad (*)$$

may serve as a warning. An unbelievably huge literature tried to establish a transcendental "law of logistic growth"; measured in appropriate units, practically all growth processes were supposed to be represented by a function of the form (*) with t representing time. Lengthy tables, complete with chi-square tests, supported this thesis for human populations, for bacterial colonies, development of railroads, etc. Both height and weight of plants and animals were found to follow the logistic law even though it is theoretically clear that these two variables cannot be subject to the same distribution. Laboratory experiments on bacteria showed that not even systematic disturbances can produce other results. Population theory relied on logistic extrapolations (even though they were demonstrably unreliable). The only trouble with the theory is that not only the logistic distribution but also the normal, the Cauchy, and other distributions can be fitted to the same material with the same or better goodness of fit [3]. In this competition the logistic distribution plays no distinguished role whatever; most contradictory theoretical models can be supported by the same observational material.

Theories of this nature are short-lived because they open no new ways, and new confirmations of the same old thing soon grow boring. But the naive reasoning as such has not been superseded by common sense, and so it may be useful to have an explicit demonstration of how misleading a mere goodness of fit can be.

2.3. Standard Validation Tasks

In order to help establish the expectation that new metrics will be evaluated against existing competitors, it would be useful to have a set of standard problems, tasks, or programs against which new metrics can (should) be validated. This role is played by sorting programs for algorithmic complexity, by the "reader-writer" problem for process synchronization, by stacks for data type specifications, by the M/M/1 queue in performance evaluation, and by the fruit fly *Drosophila melanogaster* for genetics. Such a problem or program can become a standard in its own right,

simply because so much knowledge grows up around it.

3. Selecting Appropriate Metrics for a Task

As software measurement is currently practiced, we can observe two unfortunate patterns. The first is a tendency to use statistics uncritically; the second is a tendency to obtain as much detail or precision as possible, without regard for the cost. We must recognize that there are legitimate grounds for departing from the singleminded pursuit of exact measurements and detailed models.

The modern computer has an unparalleled capacity for producing data. The problem in applying software metrics is to find appropriate measures and make sense out of the data, not simply to obtain the data. It is important to bear in mind that collecting and analyzing statistics itself has cost, and that many parts of a system may require analysis. When much of a system is not analyzed, or only crudely analyzed, there may be little merit in going to great lengths to obtain exact measurements of individual parts.

Imprecise measures are often good enough, and they should be cheap to compute. They are particularly likely to be adequate in large systems where parts of the system other than the one being measured are even less well analyzed or understood.

A user cannot make reasonable decisions about the selection of a metric without data on both the cost and the precision of the metric. Thus the description of a metric should indicate how much of what kind of data is required and how expensive it is likely to be to collect the data and compute the measurement. In addition, the description of the metric should specify the precision, reliability, and validity of the results.

This point can be supported by citing a few specific examples:

Card et al [1] needed a model for predicting the time it would take for expert users to execute interactive tasks on an interactive system. They developed a model, the Keystroke Model, based solely on counting keystrokes and other low-level operations. Validation showed a root-mean-square prediction error of 21% for individual tasks. In addition, the model can be simplified in ways that directly trade ease-of-use for accuracy.

Wulf and Feiler [15] have developed a method for cheaply determining a figure of merit that estimates the improvement made by an optimizing compiler for various <compiler, machine> combinations. The figure of merit represents the code size of the optimized code, scaled to an arbitrary standard. It relies on an assumption about hierarchy and independence among optimization techniques, and it is limited to algebraic languages and conventional register-oriented machines. Within those limitations, the figure-of-merit for a new <compiler, machine> combination can be

obtained by compiling and analyzing a small set of test programs; the time required is estimated to be a day or two. Initial validation studies across several languages and machines indicate that the figures of merit can be about 90% accurate (10% confidence level).

The "Fog Index" is a measure of the syntactic complexity of English text. Proposed by Gunning [7], it relies on counts of sentence lengths and the density of polysyllabic words to obtain an estimate of the number of years of education a reader would need in order to read a piece of text. The index can clearly be defeated, for it gives no consideration to the conceptual difficulty of the material, only to the sentence structure. Nevertheless, it proves a useful indicator to identify problem areas in written material. It is of interest here because it delivers a great deal of information in proportion to the (very small) amount of effort required to compute it.

A graphic demonstration of the tradeoff between precision and analysis cost is provided by Euclid's Algorithm for finding the greatest common divisor of two numbers, m and n (say $n > m$). Knuth [10] (sec 4.5.3, pp.316-333) devotes seventeen pages to an empirical argument that in the worst case the number of divisions required is approximately $1.9405 \log_{10} n$. However, the observation that each divisor is at least two, so the larger of the numbers under consideration must be halved at each step leads immediately to a bound of $2 \log_2 n$, or $6.0206 \log_{10} n$. If it is sufficient for your purposes to have a bound that is only within a factor of at out 3, the analysis to support the bound is much simpler and intuitively plausible.

4. Summary

In this note I have sketched some issues concerning the evaluation and selection of software metrics. A number of questions remain open for investigation:

- Can we identify a reasonably small collection of techniques that satisfy a large fraction of software measurement needs? It would be unfortunate if we needed to devise new approaches for most new problems.
- We don't currently know which properties of a metric matter or how high we can reasonably expect correlations, reliabilities, or other indicators to be. We have lots of experience with particular models, but no good generalizations. How can we encourage the process of generalization?
- How can we best draw on traditional statistics? In particular, how do we (a) transfer knowledge from statistical fields; (b) educate computer scientists in statistical responsibilities; (c) determine whether techniques can be taken directly from statistical fields or whether they must be modified for computer science?
- Do traditional techniques deal with all of our problems, or do we need new techniques to cope with (a) the ease with which we generate large masses of data; (b) the fact that we are still developing our models; (c) the apparent malleability of our medium and the ease with which we can blend models with running systems?

- Recent developments in programming methodology have emphasized the use of abstraction and hierarchical organization to control detail. How can these program organizations be exploited by the metrician? apply here?
- Performance evaluation techniques now address hardware problems quite successfully. Are there significant differences between hardware and software problems, or do we just have more experience with measuring hardware?
- Tukey [14] makes the point that conventional statistical techniques are useful for testing hypotheses we have already formulated. Is software metrics now in such a state that we need assistance in formulating the hypotheses more than we need help with testing them? If so, we should be cautious about expecting conventional statistics to be applicable to all our problems.

It is important to note that understanding is much more important than tools. Although measurements may help us come to understand them, in the long run comprehension of the underlying processes is critical to useful metrics.

5. References

1. Stuart K. Card, Thomas P. Moran, Allen Newell. The Keystroke-Level Model for User Performance Time with Interactive Systems. Tech. Rept. SSL-79-1, Xerox PARC, March, 1979.
2. Data and Analysis Center for Software. Quantitative Software Models. U. S. Air Force, March, 1979.
3. William Feller. "On the logistic law of growth and its empirical verifications in biology." *Acta Biotheoretica* 5 (1940), 51-66.
4. William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley and Sons, 1956.
5. John E. Freund. *Mathematical Statistics*. Prentice-Hall, 1971.
6. Edwin E. Ghiselli. *Theory of Psychological Measurement*. McGraw-Hill, 1934.
7. Robert Gunning. *How to Take the Fog Out of Writing*. Dartnell Press, 1964.
8. Paul Horst. *Psychological Measurement and Prediction*. Wadsworth, 1966.
9. L. Kleinrock. *Queuing Systems, Volume I: Theory*. John Wiley & Sons, 1975.
10. Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1969.
11. Teresa Lynn Roberts. *Evaluation of Computer Text Editors*. Ph.D. Th., Stanford University, 1980.
12. Jeffrey R. Spirn. *Program Behavior: Models and Measurements*. Elsevier Scientific Publishing Company, 1977.

13. Liba Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*. Elsevier Scientific Publishing Company, 1976.
14. John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.
15. W. Wulf, P. Feiler, J. Zinnikas, R. Brender. A Quantitative Technique For Comparing the Quality of Language Implementations. In preparation.

Annotated Bibliography on Software Metrics

Prepared for ONR Study Panel on Software Metrics

Compiled by Mary Shaw
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

12 June 1980

This bibliography includes the significant references from the papers in the panel report together with other papers that have been identified as relevant to some aspect of software metrics. It is interesting to note that almost all of the papers were published within the past ten years.

Contributions of panel members and other interested members of the program measurement community were augmented with authors' abstracts and reviews from *Computing Reviews*. The source of each review is indicated in the annotation. No attempt has been made to make the bibliography complete -- the boundaries of the field are too fuzzy and the number of projects is too large. Nevertheless, the annotations of provided here should serve as a useful resource.

The format of the bibliography is patterned on a software engineering bibliography developed at the University of Toronto in the mid-1970's [Barnard77]. The first section lists the papers on each of several topics, and the second section provides complete citations and annotations.

The bibliography is available in Scribe format on the ARPANet. Contact Shaw@CMU-10A for details.

Authors of Annotations

Attribution	Author and affiliation
Abstract	Abstract of the cited paper
CR12345	Computing Reviews (review number included)
JES	Jean Sammet, IBM Corporation
LAB	L. A. Belady, IBM Corporation
MS	Mary Shaw, Carnegie-Mellon University
RTS	Richard T. Snodgrass, Carnegie-Mellon University
VRB	V. R. Basili, University of Maryland

Topic Lists

General

Basili78c, Beilner77, Belady79a, Boehm73, Chen76, Gepner78, Gilb77c, QSM79, Mohanty79, Schneiderman80, Spirn77, Svobodova76

Surveys and Bibliographies

Atwood79, Barnard77, Basili78c, Belady79a, Calingaert67, Conti79, DACS79, Ferrari78, Fitzsimmons78, Gilb77a, Jackson78, QSM79, Mohanty79, Schneiderman80

Case Studies

Baker77, Basili75, Basili78a, Basili79b, Boisvert79, BrinchHansen73, Browne70, Card78, Clark76, Clark78, Clark79, Freberger79, Gannon75a, Gannon75b, Gannon77, Gehring77, Griswold75, Gupta78, Hansen78, Jackson78, Myser78b, Rye77, Sammet71, Schneiderman80, Weiss79, Willman77, Woodfield79

Models

Basili78c, Beilner77, DACS79, Gilb77c, Halstead77, QSM79

Modeling Concerns: Basili77, Basili78a, Card78, Card79a, Card79b, Courtois77, Gehring77, Lyness79, Mills76, Musa75, Naur78, Roberts80, Sevcik74, Smith79b, Spirn77, Svobodova76, Tuggle78

Structural: Bard79, Basili77, Basili78a, Belady79b, Boehm73, Booth79, Chanon73, Courtois77, Denning78, Ellis78, Gupta78, Howden78a, Jones78, Kleinrock75a, Lyness79, McCabe76, Musa75, Parr80, Putnam78, Sevcik74, Shaw74, Shaw79, Trivedi79, Walters78, Waters79, Wulf??

Empirical: Basili77, Basili78a, Basili78d, Belady76, Cheng79, Cheung74, Chrysler78, Dunsmore78, Elshoff76, Fitsos79, Fitsos80, Freberger79, Goel78, Howden78a, Jeffery??, Jones78, Laemmel78, Motley77, Myers78a, Schafer79, Smith79a, Smith80, vanderKnijff78, Walston77, Walters78, Weiss79, Weissman73, Weissman74, Wolverton74, Woodfield79

Evaluation of Metrics

Basili78b, Basili78c, Basili79a, Bloom74, Boehm73, Boisvert79, Budd78, Curtis80, deFreitas78, Elshoff76, Gehring77, Gordon79b, Halstead77, Howden78a, Howden78b, Howden79, Jackson78, QSM79, Lyness79, Mamrak79, Sammet71, Schneiderman80, Sevcik74, Tuggle78

Direct Measurement of Programs

Belady79a, Schneiderman80

Static: Alexander72, Baker79, Basili75, Basili77, Basili78b, Basili79a, Basili79c, Belady79b, Brailsford77, Chanon73, Curtis79, Dunsmore80, Elshoff76, Fitzsimmons78, Foxley78, Gannon75a, Gannon75b, Gannon77, Gordon79a, Halstead77, Jeffery??, Knuth71, Laemmel78, Love77, McCabe76, Ryder79, Shaw74, Slavinski75, vanderKnijff78, Wichmann70, Woodward79, Zweben79

Dynamic: Ashby73, Barak78, Beizer70, Bergeron75, Brailsford77, BrinchHansen73, BrinchHansen78, Calingaert67, Cambell68, Cerf70, Cheng69, Cohen74, Cohen77, Crochow69, Crowley79, Dearnley78, Elshoff76, Fitch77, Fong73, Gaines69, Griswold75, Hanson78, Ingalls71, Johnston70, Knuth71, Knuth73, Lyon75, Matwin76, Millbrant74, Model78, Perrott77, Russell69, Saltzer70, Satterthwaite72, Sites78, Storey77, Waite73, Wichmann70, Wong74, Wortman76, Yuval75

Performance Evaluation

Bard76, Bard79, Beilner77, Booth79, Calingaert67, Campbell68, Conti79, Denning78, Ferrari78, Kleinrock75b, Robinson79, Smith79b, Storey77, Trivedi79

Data Collection Techniques

Baker77, Basili75, Basili77, Basili78b, Basili78c, Basili79a, Basili79c, Bergeron75, Cheung74, Cohen74, Ferrari78, Fries76, Saltzer70, Schneiderman80, Slavinski75, Walston77, Willman77

Human Factors

Atwood79, Basili78b, Basili79c, Card78, Card79a, Card79b, Dunsmore80, Gannon75a, Gannon75b, Gannon77, Gordon79a, Hansen78, Love77, Model78, Roberts80, Sammet71, Schneiderman80, Sheppard79, Weissman73, Weissman74, Woodfield79

Error Studies

Amory75, Basili77, Basili78a, Basili78b, Basili78d, Basili79b, Belady76, Dunsmore78, Dunsmore80, Fries76, Gannon75a, Gannon75b, Gannon77, Motley77, Musa75, Schafer79, Schneiderman80, Thayer76, Weiss79

Measures of Productivity

Chrysler78, Comer79, Crossman79, Curtis79, Dunsmore78, Fitzsimmons78, Freberger79, Jeffery??, Jones78, Myers78a, Sammet70, Sheppard79, Walston77, Woodfield79

Software Life Cycle

Basili78c, Basili78d, Dunsmore80, Gehring77, Gilb77b, Hoare76, Jeffery??, Lehman80, McKissick79, Mills76, Myers78a, Parr80, Putnam78, Rye77, Weiss79, Wolverton74

Cost Estimation

Basili77, Basili78a, Basili79b, Belady76, DACS79, Doty77, Herd77, Jeffery??, Jones78, Parr80, Putnam78, Walston77, Wolverton74

Software Monitoring

Ashby73, Barak78, Basili75, Basili77, Basili78b, Basili79a, Basili79c, Beizer70, Bergeron75, Brailsford77, BrinchHansen73, BrinchHansen78, Calingaert67, Cambell68, Cerf70, Cheng69, Cohen74, Cohen77, Crochow69, Crowley79, deFreitas78, Dearnley78, Elshoff76, Fitch77, Fong73, Gaines69, Griswold75, Hanson78, Ingalls71, Johnston70, Knuth71, Knuth73, Lyon75, Matwin76, Millbrant74, Model78, Perrott77, Russell69, Saltzer70, Satterthwaite72, Sites78, Storey77, Waite73, Wichmann70, Wong74, Wortman76, Yuval75

Transfer of Techniques from Other Disciplines

McCabe76, Musa75, Schneiderman80

Maintenance and Enhancement

Belady76, Curtis79, Gilb77b, McKissick79, Walters78

Language Selection, Evaluation, and Design

Browne70, Dunsmore78, Gannon75a, Gannon75b, Gannon77, Hoare76, Sammet71, Shaw74, Shaw80, Weissman73, Weissman74

Application-Specific Measures

Compilers: Bloom74, Shaw74, Wulf??

Interactive Systems: Card78, Card79a, Card79b, Mamrack79, Roberts80

Paging Systems: Arora78, Gupta78, Sprin77

Protection: Ellis78

Reliability: Gilb77a, Gilb77b, Littlewood75, Musa75, Naur78, Schafer79, Thayer76, Walters78

Testing Techniques: Bicevskis79, BrinchHansen73, BrinchHansen78, Budd78, Howden78a, Howden78b, Howden79, Myers78b

Annotated Bibliography

- [Alexander 72] William Gregg Alexander.
How A Programming Language is Used.
 Technical Report CSRG-10, University of Toronto, February, 1972.
 An empirical study of programs written in XPL was carried out with the aim of determining properties of both the language and the object machine, the S/360. The information gathered by examining a set of typical XPL, to the compiler for XPL, and to the S/360. In addition, a profile generator for XPL programs was developed and illustrated. A powerful programming tool, it enables an XPL programmer to clearly see where execution time is spent in his program. [Abstract]
- [Amory 75] W. Amory and J. A. Clapp.
Engineering of Quality Software Systems (A Software Error Classification Methodology).
 Technical Report RADC-TR-74-325, Rome Air Development Center Technical Report, January, 1975.
 This report presents preliminary results of a study in the area of error classification. A general method of error classification is described which is designed to serve as a guideline for experiment-specific application. A survey of error classification and analysis work, both in the general literature and at MITRE, as well as a study of error experiment design considerations, are reflected in the discussion and conclusions. [Abstract]
- [Arora 78] Radha Krishan Arora and R. K. Subramanian.
 Exploiting the optimal paging algorithms.
Inf. Process. Lett. 7(5):233-236, August, 1978.
 From the title of this paper, I expected a discussion of a way to improve existing paging techniques by applying new insights about optimal paging algorithms. What I found instead was a presentation of methods for computing: 1) the number of page faults and the average memory demand for a variable space page replacement algorithm which minimizes the total cost of replacement and retention of pages; and 2) the minimum number of page faults for a fixed space demand prepaging algorithm. Although these paging algorithms are unrealizable (because they require knowledge of future references), they are often useful as benchmarks for comparisons with realizable replacement algorithms. The methods for computing these values appear to be quite efficient, requiring only one pass over the reference trace and one memory cell per program page. The paper explains the methods clearly, and I would recommend it to any researcher desiring to compute these performance measures. [I:R34149]
- [Ashby 73] Gordon Ashby, Loren Salmonson, and Robert Heilman.
 Design of an Interactive Debugger for FORTRAN: MANTIS.
Software--Practice and Experience 3:65-74, 1973.
 A debugger for FORTRAN is described. Actions can be associated with individual statements. The execution flow can also be traced. Objectives, implementation experience and system features are discussed and are related to the general problem of designing debuggers for language subsystems. [RTS]

[Atwood 79]

Michael E. Atwood, H. Rudy Ramsey, Jean N. Hooper and Daniel A. Kullas.
Annotated Bibliography on Human Factors in Software Development.

AIR Technical Report P-79-1, June, 1979.

As part of a larger Army Research Institute effort to survey, synthesize, and evaluate the state of the art in the area of human factors as applied to software development, a fairly extensive literature survey was conducted. This resulting bibliography contains citations of 478 articles or reports pertaining to the behavioral aspects of software design, programming, coding, debugging, testing, evaluation, and maintenance. Most citations are accompanied by descriptive abstracts, and all are indexed by author, publication source, institutional affiliation, and subject. To help the user unfamiliar with the area, the bibliography contains brief, basic reference lists in the areas of software engineering, the psychology of software development, the Structured Programming Series, and the DoD software program. Coverage is exhaustive through 1977 with a few references in 1978. [Abstract]

[Baker 77]

W. F. Baker.

Software Data Collection and Analysis: A Real-Time System Project History.

Technical Report RADC-TR-77-192. Rome Air Development Center, June, 1977.

This report discusses the procedures used for, and the results obtained from, an analysis of software error problem reports. The problem reports studied were generated during the development of a large, real-time, highly sophisticated multi-processor data processing system. A brief profile of the development of this system is presented along with discussions of the procedures used in the analysis of the problem reports and the objectives to be met. The results of the analysis are discussed, and statistics reflecting the results are presented. Finally, some of the problems encountered during the course of the study are presented, as well as some pertinent observations. [Abstract]

[Baker 79]

Albert L. Baker and Stuart H. Zweben.

The Use of Software Science in Evaluating Modularity Concepts.

IEEE Transactions on Software Engineering SE-5(2):110-120, March, 1979.

An investigation is made into the extent to which relationships from software science are useful in analyzing programming methodology principles that are concerned with modularity. Using previously published data from over 500 programs, it is shown that the software science effort measure provides quantitative answers to questions concerning the conditions under which modularization is beneficial. Among the issues discussed are the reduction of similar code sequences by temporary variable and subprogram definition, and the use of global variables. Using data flow analysis, environmental consideration which affect the applicability of alternative modularity techniques are also discussed.

The results obtained using software science are compared with certain generally accepted methodologies involving modularity, and show strong agreement. Finally, the results suggest some areas of potential improvement in the technique used to obtain the software science measurements. [Abstract]

[Barak 78]

Amnon B. Barak and Moshe Aharoni.

A Study of Machine Level Software Profiles.

Software--Practice and Experience 8:131-136, 1978.

The instruction mix of a CDC CYBER/74 computer in a university environment was monitored, and frequencies of execution for the most commonly used instructions was determined. The percentages over various time intervals is constant, so a machine-level software profile (MLSP) can be computed. [RTS]

[Bard 76]

Yonathan Bard.

An Experimental Approach to System Tuning.

In Peter P. S. Chen and Mark Franklin, editor, *Proceedings of the International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 296-305. Harvard University, March, 1976.

It is desired to find the values of certain system parameters which maximize some performance criterion. Using standard experimental design techniques, one runs an initial set of experiments which explore the system's response surface. Subsequently, the data are smoothed, and a hill climbing technique is used to locate the maximum. This technique was employed successfully to tune the parameters in an experimental version of the VM/370 scheduler. [Abstract]

The techniques of measurement described here have the great advantage of neither pretending to specious accuracy nor consuming the resources they are supposed to measure during the experimental period. This would seem a method with wide applicability in tuning systems, and one capable of useful refinement with experience. [CR34690]

[Bard 79]

Yonathan Bard.

Performance Analysis of Interactive Systems.

In *Quantitative Software Models*, pages 108-124. Data and Analysis Center for Software, 1979.

Many topics are covered by the science (art??) of system performance analysis. Among these are measurement of existing systems, tuning, performance evaluation, design of control algorithms, system modeling, workload characterization, and performance prediction. After giving a brief summary of these topics, the paper will concentrate on the last three. It will describe how existing workloads can be measured and analyzed routinely so as to produce the inputs required by analytic system models. These models employ queueing network formulations which allow the workload to consist of several user classes with different characteristics. Such models have been validated successfully against real systems, but some problems, particularly relating to paging in virtual memory systems, have so far not received definitive solutions. [CR34534]

[Barnard 77]

David Barnard (ed.).

An Annotated Bibliography on Computer Program Engineering.

Technical Report, University of Toronto, Computer Systems Research Group, May, 1977.

Collects extensive annotations on individual articles in the field of software engineering and includes subject classification and extensive topic cross-referencing. Many of the referenced articles are relevant to software metrics. [MS]

[Basili 75]

V. R. Basili and A. J. Turner.

Iterative Enhancement: A Practical Technique for Software Development.

IEEE Transactions on Software Engineering SE-1(4), December, 1975.

This paper recommends the "iterative enhancement" technique as a practical means of using a top-down, stepwise refinement approach to software development. This technique begins with a simple initial implementation of a properly chosen (skeletal) subproject which is followed by the gradual enhancement of successive implementations in order to build the full implementation. The development and quantitative analysis of a production compiler for the language SIMPL-T is used to demonstrate that the application of iterative enhancement to software development is practical and efficient, encourages the generation of an easily modifiable product, and facilitates reliability. [VRB]

[Basili 77]

Victor R. Basili, Marvin V. Zelkowitz, Frank E. McGarry, Robert W. Reiter, Walter F. Truszkowski and David L. Weiss.

The Software Engineering Laboratory.

Technical Report TR-535, University of Maryland, Computer Science Center, College Park, Maryland, May, 1977.

The development of techniques to produce cost-effective reliable software first requires the collection of quantitative and qualitative data on the development process. Towards this end, the Software Engineering Laboratory has been organized in conjunction with NASA Goddard Space Flight Center. The purpose of the Software Engineering Laboratory is to monitor existing software methodologies and develop and measure the effectiveness of alternate methodologies.

Initially, three aspects of the software development life cycle are to be investigated. These are: (1) management aspects in estimating team organization, resource requirements, schedules and reliability factors in the finished software product, (2) error characteristics and their causes, and (3) program structure and its relation to well-developed software. [VRB]

[Basili 78a]

Victor R. Basili and Marvin V. Zelkowitz.

Analyzing Medium-Scale Software Development.

In *Proceedings of the Third International Conference on Software*

Engineering, IEEE Catalog No. 78CH1317-7C, Atlanta, Georgia, May, 1978.

The collection and analysis of data from programming projects is necessary for the appropriate evaluation of software engineering methodologies. Towards this end, the Software Engineering Laboratory was organized between the University of Maryland and NASA Goddard Space Flight Center. This paper describes the structure of the Laboratory and provides some data on project evaluation from some of the early projects that have been monitored. The analysis relates to resource forecasting using a model of the project life cycle based upon the Rayleigh equation and to error rates applying ideas developed by Belady and Lehman. [VRB]

[Basili 78b]

Victor R. Basili and Robert W. Reiter, Jr.

Investigating Software Development Approaches.

Technical Report TR-688, University of Maryland, Department of Computer Science, August, 1978.

This paper reports on research comparing various approaches, or methodologies, for software development. The study focuses on the quantitative analysis of the application of certain methodologies in an experimental environment, in order to further understand their effects and better demonstrate their advantages in a controlled environment. A series of statistical experiments were conducted comparing programming teams that used a disciplined methodology (consisting of top-down design, process design language usage, structured programming, code reading, and chief programmer team organization) with programming teams and individual programmers that employed ad hoc approaches. Specific details of the experimental setting, the investigative approach (used to plan, execute, and analyze the experiments), and some of the results of the experiments are discussed. [VRB]

[Basili 78c]

Victor R. Basili, Edward H. Ely and Donovan Young.

Executive Summary of the Second Software Life Cycle Management Workshop.

Proceedings of Second Software Life Cycle Management Workshop.

Workshop held at Atlanta, Georgia in August, 1978.

This is the proceedings of a workshop sponsored by the U. S. Army Computer Systems Command, U. S. Army Institute for Research in Management Information and Computer Science. It contains a large volume of papers which deal with models, metrics, and studies in the software life cycle management process. Topics of interest include (1) description and understanding of various components of the life cycle, (2) ways to delineate and analyze relationships among component activities, (3) milestones and other tools to help direct, coordinate, understand and control research and development in software life cycle management, and (4) development of management tools using the

results of life cycle management research to help plan and manage software development projects. [VRB]

[Basili 78d]

Victor R. Basili and Marvin V. Zelkowitz.

Analyzing Medium-Scale Software Development.

Proceedings of the Third International Conference on Software Engineering:116-123, May, 1978.

The collection and analysis of data from programming projects is necessary for the appropriate evaluation of software engineering methodologies. Towards this end, the Software Engineering Laboratory was organized between the University of Maryland and NASA Goddard Space Flight Center. This paper describes the structure of the Laboratory and provides some data on project evaluation from some of the early projects that have been monitored. The analysis relates to resource forecasting using a model of the project life cycle based upon the Rayleigh equation and to error rates applying ideas developed by Belady and Lehman. [Abstract]

[Basili 79a]

Victor R. Basili and Robert W. Reiter, Jr.

Evaluating Automatable Measures of Software Development.

IEEE Workshop on Quantitative Software Models, October, 1979.

There is a need for distinguishing a set of useful automatable measures of the software development process and product. Measures are considered *useful* if they are sensitive to externally observable differences in development environments and their relative values correspond to some intuition regarding these characteristic differences. Such measures could provide an objective quantitative foundation for constructing quality assurance standards and for calibrating mathematical models of software reliability and resource estimation. This paper presents a set of automatable measures that were implemented, evaluated in a controlled experiment, and found to satisfy these usefulness criteria. The measures include computer job steps, program changes, program size, and cyclomatic complexity. [VRB]

[Basili 79b]

Victor R. Basili and Marvin V. Zelkowitz.

Measuring Software Development Characteristics in the Local Environment.

Computers and Structures 10, 1979.

This paper discusses the characterization and analysis facilities being performed by the Software Engineering Laboratory which can be done with minimal effort on many projects. Some examples are given of the kinds of analyses that can be done to aid in managing, understanding and characterizing the development of software in a production environment. [VRB]

[Basili 79c]

Victor R. Basili and Robert W. Reiter, Jr.

An Investigation of Human Factors in Software Development.

Computer Magazine, December, 1979.

This paper gives a human factors interpretation of an experiment on software development. The experiment involves the development of software under three different environments which include disciplined teams, ad hoc teams, and ad hoc individuals. Low level programming aspects are used to predict high level software properties, such as reliability, cost effectiveness, and complexity. [VRB]

[Beilner 77]

H. Beilner and E. Gelenbe.

Measuring, modeling and evaluating computer systems.

Elsevier/North Holland, Inc., New York, 1977.

This book consists of the 26 papers presented at the Third International Symposium on Modeling and Performance Evaluation of Computer Systems organized in Bonn, W. Germany (October 1977) by GMD and cosponsored by IFIP Working Group 7.3, IRIA-LABORIA, and the Commission of the European Communities.

Five contributions aim at the evaluation of more or less total computer system configurations; four papers deal with models of program behavior and of memory management policies; seven others propose exact or approximate solutions for probabilistic models of queues and/or of networks of queues. There are two studies of

optimal resource assignment under deadlock and cost constraints, respectively. Two papers discuss the use of timed Petri nets for performance evaluation. The remaining presentations deal with simulation models (2), measurement tools (1), statistical sequential methods (1), and reliability assessment of fault-tolerant computing systems (1).

Most papers are contributions to research; this book is therefore not for the manager in search of a ready-made method for the evaluation of his computer installation.

The book ends with the keynote address which was delivered by C. A. Petri. This is a rather controversial and thought-provoking paper in which it is argued that the existence and impact of computer technology on society "makes a new approach to modeling necessary and at the same time feasible." While the necessity is evident to many people working in the field, the feasibility is less so, and the two examples by which Petri illustrates the new "strict modeling discipline" he proposes are unfortunately too modest to be really convincing in this respect. [CR34328]

[Beizer 70]

B. Beizer.

Analytical Techniques for the Statistical Evaluation of Program Running Times.
In *Proceedings of the Fall Joint Computer Conference*, AFIPS Press, 1970.

[Belady 76]

L. A. Belady and M. M. Lehman.

A Model of Large Program Development.
IBM Systems Journal(3), 1976.

Discussed are observations made on the development of OS/360 and its subsequent enhancements and releases. Some modeling approaches to organizing these observations are also presented. [VRB]

[Belady 79a]

L. A. Belady.

Survey of software complexity measures.

Proceedings of the IEEE Workshop on Quantitative Software Models.

Workshop held at Kiamesha Lake, New York, in October, 1979.

The survey is presented in a two-dimensional classification of about seventy papers on complexity. Both programming related and more general papers are considered. Following the approach taken by a paper, the survey distinguishes four classes: informal, based on counting, probabilistic, and experimental. Some examples are offered. [LAB]

[Belady 79b]

L. A. Belady and C. J. Evangelisti.

System Partitioning and its Measure.

Technical Report RC 7560 (# 32643), IBM T. J. Watson Research Center, March, 1979.

Program modules and data structures are interconnected by calls and references in software systems. Partitioning these entities into clusters reduces complexity. For very large systems manual clustering is impractical. A method to perform automatic clustering is described and a metric to quantify the complexity of the resulting partition is developed. [Abstract]

[Bergeron 75]

R. Daniel Bergeron and Henri Bulterman.

A Technique for Evaluation of User Systems on an IBM S/370.

Software--Practice and Experience 5:83-92, 1975.

The design and implementation of the System for System Development (SSD) Evaluation System is described. This system modifies load modules of the user's system in order to cause run-time invocation of the system at routine entry. Raw data on secondary storage is produced, and a post processor is used to interpret the data. The information produced by the post processor includes cumulative and differential times, execution counts, and a history of the calling sequences. [RTS]

- [Bicevskis 79] Janis Bicevskis, Juris Borzovs, Uldis Straujums, Andris Zarins, and Edward F. Miller, Jr.
 SMOTL -- A system to Construct Samples for Data Processing Program Debugging.
IEEE Transactions on Software Engineering SE-5(1):60-66, January, 1979.
 The possibility of automatic construction of a complete set of program tests is considered. A test set system is said to be complete if every feasible program branch (segment) is executed by it. The complete test set construction algorithm for commercially oriented data processing programs is outlined, and the results of its functioning on real programs are analyzed. [Abstract]
- [Bloom 74] Burton H. Bloom, Mac H. Clark, Clare G. Feldman, Robert K. Coe.
 Criteria for Evaluating the Performance of Compilers.
 Technical Report RADC-TR-74-259, Rome Air Development Center, October, 1974.
 The main purpose of this study was to develop criteria by which it will be possible to qualitatively measure and evaluate the performance of compilers, possibly operating on different computers, and possibly having different features. To satisfy this purpose, three technical questions were studied:
1. How can two compilers with the same features and operating in the same environment be compared?
 2. If two compilers with the same features operate in different environments, how can their measured differences in performance be attributed to the environmental differences vs. the compiler differences?
 3. How should a compiler buyer deal with the problem of evaluating compilers with different special features?
- These three questions were studied from a point of view that the answers should help provide a basis for conducting dollar cost/benefit analysis of compilers. [Abstract]
- [Boehm 73] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. McLeod, M. J. Merritt.
 Characteristics of Software Quality.
 Technical Report TRW-SS-73-09, TRW Software Series Report, December, 1973.
 This may have come out as a Springer or Elsevier book.
 The objectives of this study were to identify a set of characteristics of quality software and, for each characteristic, to define a metric such that:
1. Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic.
 2. Overall software quality can be defined as some function of the values of the metrics.
- [Abstract]
- [Boisvert 79] Ronald F. Boisvert, John R. Rice and Elias N. Houstis.
 A System for Performance Evaluation of Partial Differential Equations Software.
IEEE Transactions on Software Engineering SE-5(4):418-425, July, 1979.
 This paper describes a system to systematically compare the performance of various methods (software modules) for the numerical solution of partial differential equations. We discuss the general nature and large size of this performance evaluation problem and the data one obtains. The system meets certain design objectives that ensure a valid experiment: 1) precise definition of a particular measurement; 2) uniformity in definition of variables entering the experiment; and 3) reproducibility of results. The ease of use of the system makes it possible to make the large sets of measurements necessary to obtain confidence in the results and its portability allows others to check or extend the measurements. The system has four parts: 1) semiautomatic generation of problems for experimental input; 2) the *ellpack* system for actually solving the equation; 3) a data management system to organize and access the experimental data; and 4) data analysis programs to extract graphical and statistical summaries from the data. [Abstract]

[Booth 79]

Taylor L. Booth.

Performance Optimization of Software Systems Processing Information Sequences
Modeled by Probabilistic Languages.

IEEE Transactions on Software Engineering SE-5(1):31-44, January, 1979.

The performance of a hardware/software system is a function of both the deterministic properties of the computation being performed and the probabilistic properties of the information sequence being processed. This paper develops the idea of a computational cost which can be used to measure system performance and shows how this cost can be related to the structure of the system and the information processed by the system. Different ways in which this cost can be used to study system performance are presented. [Abstract]

[Brailsford 77]

D.F. Brailsford, E. Foxley, K.L. Mander and D.J. Morgan.

Run-Time Profiling of Algol 68-R programs using DIDYMUS and SCAMP.

SIGPlan Notices 12(6):27-35, June, 1977.

Proceedings of the Strathclyde Algol 68 Conference.

Two programs are discussed. The first, DIDYMUS, is run together with the program to be monitored, and builds a histogram of the program counter at regular intervals. A postprocessor which has access to the link table then prints the approximate time spent in each routine. The second program, SCAMP, is a preprocessor which inserts code to maintain counts of syntactic constructs. Since only a moderate amount of syntactic analysis is done, there are several restrictions on the allowable constructs. [RTS]

[BrinchHansen 73]

Per Brinch Hansen.

Testing a Multiprogramming System.

Software--Practice and Experience 3:145-150, 1973.

A central problem in program design is to structure a large program such that it can be tested systematically by the simplest possible techniques. This paper describes the method used to test the RC 4000 multiprogramming system. During testing, the system records all transitions of processes and messages between various queues. The test mechanism consists of fifty machine instructions centralized in two procedures. By using this mechanism in a series of carefully selected test cases, the system was made virtually free error free within a few weeks. The test procedure is illustrated by examples. [Abstract]

[BrinchHansen 78]

Per Brinch Hansen.

Reproducible Testing of Monitors.

Software--Practice and Experience 8:721-729, 1978.

This paper describes a systematic method for testing monitor modules which control process interactions in concurrent programs. A monitor is tested by executing a concurrent program in which the processes are synchronized by a clock to make the sequence of interactions reproducible. The method separates the construction and implementation of test cases and makes the analysis of a concurrent experiment similar to the analysis of a sequential program. The implementation of a test program is almost mechanical. The method, which is illustrated by an example, has been successfully used to test a multicomputer network program written in Concurrent Pascal. [Abstract]

[Browne 70]

P. H. Browne et al.

Data Processing Technologies, Volume I - High-Level Language Evaluation.

Army Contract No. DAH60-69-C-0037, Teledyne Brown Engineering, Huntsville, Alabama, May, 1970.

This paper discusses the evaluation criteria used for selecting languages for use in Ballistic Missile Defense Agency. While most of the paper is devoted to a discussion of the relevance of the technical characteristics of the languages under consideration, the report is important for software metrics because it describes a weighted scoring technique and applies this to a practical situation. [JES]

[Budd 78]

Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, and Richard A. DeMillo.

The Design of a Prototype Mutation System for Programming Testing.

In *Proc. 1978 National Computer Conference*, pages 623-628. AFIPS, November, 1978.

One of the most neglected areas of software development has been the area of software testing. There have, however, been many articles written on the inadequacy of the testing performed on software prior to release.

The authors are correct when they state that the "major question which must always be addressed is: If a program is correct for a finite number of test cases, can we assume it is correct in general."

We are presented with a "design of a prototype mutation system for program testing." This analysis is based upon the assumption that "competent programmers will produce programs which, if they are not correct, are 'almost' correct. That is, if a program is not correct it is a 'mutant'--it differs from a correct program by simple errors." Mutation analysis is designed to detect these errors. The prototype system is approximately 80 percent effective using 25 mutant operators. These operators range from simple to complex.

If the procedures discussed are effective on subroutines, the question arises as to whether they can be as effective on larger, more complete programs. In any case, it is gratifying to see that more attention is being paid to testing software prior to release rather than waiting to correct errors detected by the user. [CR34263]

[Calingaert 67]

Peter Calingaert.

System Performance Evaluation: Survey and Appraisal.

Communications of the ACM 10(1):12-18, 1967.

The state of the art of system performance evaluation is reviewed and evaluation goals and problems are examined. Throughput, turnaround, and availability are defined as fundamental measures of performance; overhead and CPU speed are placed in perspective. The appropriateness of instruction mixes, kernels, simulators and other tools is discussed, as well as pitfalls which may be encountered when using them.

[Abstract]

[Cambell 68]

D.J. Cambell and U.J. Heffner.

Measurement and analysis of large operating systems during system development.

In *Proceeding AFIPS Fall Joint Computer Conference*, pages 903-914. AFIPS Press, 1968.

[Card 78]

Stuart K. Card.

Studies in the Psychology of Computer Text Editing Systems.

Technical Report SSL-78-1, Xerox PARC, August, 1978.

Six studies of user interaction with on-line computer text editing systems are reported. Editing times for benchmark tasks on several editing systems were collected to gauge the range of performance across systems. Using the measured times, it was possible to predict when an editing system would outperform a typewriter. To model the user's behavior in greater detail, an information processing model of editing performance is proposed describing the user's "goals", "operators", "methods", and "selection rules". An important issue in such a model is how the model's accuracy depends on the

grain of analysis. To find out, the model was recast at nine different levels of grain size and the accuracy of the different versions compared. From observations of users on several different systems, it was discovered that on each task, the users go through a similar sequence of task assimilation, target location, target modification, and verification. This concept of a "unit task cycle" was used to predict rough performance times for a proposed system prior to system specification. With respect to the target location part of a task, four devices for pointing to a target were compared and modeled. Using Fitts' Law, it is argued that the time for the best of these devices, the mouse, approaches the theoretical minimum. Finally, a Monte Carlo simulation model using gamma-distributed, with which sequences of user actions, time per task, and the distribution of time can be predicted.

The picture of user behavior that emerges from these studies is related to, but distinct from, behavior in classical problem-solving studies. The main difference is that the methods are almost certain of success. For any subproblem the user simply recalls the solution from his experience rather than working it out. Hence there is no search. Such behavior is expected to be found in many cognitive tasks in industrial work and daily life which people perform repetitively, tasks the report calls 'routine cognitive skills.' [Abstract]

[Card 79a]

Stuart K. Card, Thomas P. Moran, Allen Newell.

The Keystroke-Level Model for User Performance Time with Interactive Systems.

Technical Report SSL-79-1, Xerox PARC, March, 1979.

It is not common practice today for system designers to deal systematically with the issues of user-computer performance. One reason is the lack of appropriate analysis tools. An easy-to-use model -- the Keystroke Model -- is proposed for predicting the time it will take expert users to execute given tasks on a system. The Model is based on counting keystrokes and other low-level operations, including the user's mental preparations and the system's responses. Methods for executing tasks are coded in terms of these operations, and standard times for the operations are then summed up to give predictions. Heuristic rules are given for predicting where mental preparations occur in the methods. Keystroke Model predictions were tested against data from 28 users, on 10 systems, and over 14 task types. The root-mean-square prediction error was 21% for individual tasks -- and much better for collections of tasks. An example illustrates how the Keystroke Model can be used to give parametric predictions and how sensitivity analysis can be used to redeem conclusions in the face of uncertain assumptions. Finally, the Keystroke Model is compared to several simpler versions, which trade ease-of-use for accuracy. [Abstract]

[Card 79b]

Stuart K. Card.

A Method for Calculating Performance Times for Users of Interactive Computing Systems.

In Proceedings of the 1979 International Conference on Cybernetics and Society, pages . Cybernetics, October, 1979.

In order to design systems which are easy and pleasant to use, designers must trade off several different factors. Yet, at present, there is little scientific basis for how to do this. This paper presents a way in which one of the factors, time to perform a task, can be calculated at design time from a simple model. The technique gives estimates accurate to about 20% of actual times required by users (measured under laboratory conditions) in a variety of different tasks and systems. Several examples of the use of the model for interface design and analysis are discussed. [Abstract]

[Cerf 70]

V.G. Cerf.

Measurement of Recursive Programs.

PhD thesis, School of Engineering and Applied Science, UCLA, 1970.

Report 70-43.

[Chanon 73]

Robert N. Chanon.

On a Measure of Program Structure.

Technical Report, Carnegie-Mellon University, 1973.

Program structure has been discussed as being an important influence on the ease with which programs can be constructed, verified, understood, and changed. Yet the notion of program structure has remained a vague and imprecisely defined concept. This thesis proposes a definition and a measure for program structure and evaluates the usefulness of the measure as a tool for determining and controlling structure in a program.

Applications of the measure require that the assumptions which objects make be precisely stated. These are defined to include assumptions about the nature and use of variables and data; conditions relating to the correct execution of the program; and assumptions about the program environment in which the text is executed. Top-down programming by stepwise refinement forms the basis for a proposed methodology that permits these assumptions to be stated as a program is constructed.

The measure uses the information theoretic concept of excess entropy -- entropy loading -- to determine the extent to which assumptions are shared. Entropy loading calculations also provide a way of comparing different decompositions of a program. Unfortunately, finding the best decompositions of all but small programs seems intractable. Consequently, several heuristics are stated that attempt to establish bounds on the growth of entropy loadings for elaborations of decompositions suggested at early stages in a development.

Several programs are developed using mechanical aids to record assumptions and compute entropy loadings. Since each development preserves assumptions at every elaboration, this information need not be deduced from program text when the program is studied or is to be modified. Entropy loading figures at each stage allow different decompositions to be compared and provide either a basis for choosing a decomposition or grounds for actually modifying the program to achieve better structure. These developments illustrate the proposed methodology and show that the measure produces results that are usually consistent with the definition of program structure as well as the informal notion of structure from the literature.

Without mechanical aids, however, applications of these techniques to practical problems would be tedious and difficult. This and other difficulties motivate further research about this important but elusive property of programs: their structure.
[Abstract]

[Chen 76]

Peter P.S. Chen and Mark Franklin.

Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation.

The International Symposium on Computer Performance Modeling, Measurement, and Evaluation was held on March 29-31, 1976, at Harvard University, Cambridge, Massachusetts. This Symposium was jointly sponsored by the SIGMETRICS group of ACM and IFIP Working Group 7.3 (Computer System Modeling).

The main purpose of the Symposium was to bring together researchers and practitioners to discuss problems in computer system performance evaluation.

The papers ranged from model building and validation to measurement techniques. Several areas of current interest such as database systems, computer networks, and computer systems control were also covered. [CR34397]

- [Cheng 69] P.S. Cheng.
Trace-driven system modeling.
IBM System Journal(4):280-289, 1969.
The method described in the paper is to run the job stream sequentially using the OS under which a trace program can be executed. System activities that pertain to the execution of a given job are collected in a trace log. Then the data in the trace log is reduced to the level of detail the system is to be simulated at. The events present in the reduced trace log are used to drive the event-based simulation. [RTS]
- [Cheung 74] R. C. Cheung, K. H. Kim, C. V. Ramamoorthy, and S. S. Reddi.
Automated Generation of Self-Metric Software.
Proc. of the Seventh Hawaii International Conference on System Sciences:149-151, 1974.
Self-metric software are programs that record their behavior statistics automatically during their execution. They are generated from existing software by inserting software measurement instruments into the program at appropriate locations after an automatic analysis by the system. In this paper, an algorithm to locate and insert the optimal (minimal cost) set of activity counters for generating the program activity profile is given. The use of the model for measuring execution time, branching characteristics, code activities, path activities, and other program statistics are discussed. The applications of these statistics in program testing, optimization, and program restructuring for virtual memory are presented. [Abstract]
- [Chrysler 78] Earl Chrysler.
Some Basic Determinants of Computer Programming Productivity.
Communications of the ACM 21(6):472-483, June, 1978.
The purpose of this research was to examine the relationship between processing characteristics of programs and experience characteristics of programmers and program development time. The ultimate objective was to develop a technique for predicting the amount of time necessary to create a computer program. The fifteen program characteristics hypothesized as being associated with an increase in programming time required are objectively measurable from preprogramming specifications. The five programmer characteristics are experience-related and are also measurable before a programming task is begun. Nine program characteristics emerged as major influences on program development time, each associated with increased program time. All five programmer characteristics were found to be related to reduced program development time. A multiple regression equation which contained one programmer characteristic and four program characteristics gave evidence of good predictive power for forecasting program development time. [CR34959]
- [Clark 76] Douglas W. Clark.
List Structure: Measurements, Algorithms, and Encodings.
Technical Report, Carnegie-Mellon University, August, 1976.
This thesis is about list structures: how they are used in practice, how they can be moved and copied efficiently, and how they can be represented by space-saving encodings. The approach taken to these subjects is mainly empirical. Measurement results are based on five large programs written in Interlisp. [Abstract]
- [Clark 78] Douglas W. Clark and Cordell C. Green.
A Note on Shared List Structure in LISP.
Inf. Process. Lett. 7(6):312-314, October, 1978.
LISP permits "shared cells", that is cells which are pointed to more than once. This paper shows that fewer than 2 1/2 percent of all cells were shared for five particular applications. With so few shared cells, various clever garbage collection strategies are practical, as discussed and referenced in the paper. [CR34260]

[Clark 79]

Douglas W. Clark.

Measurements of Dynamic List Structure Use in Lisp.

IEEE Transactions on Software Engineering SE-5(1):51-59, January, 1979.

This paper is an empirical study of how three large Lisp programs use their list structure during execution. Most list-cell references are due to the functions *car* and *cdr*, which are executed about equally often and greatly outnumber other primitive functions. Executions of *cdr* yield the atom *nil* about 10 to 20 percent of the time, and nearby list cells most of the rest of the time. Executions of *car* yield atoms, small integers, and list cells in varying proportions in the three programs. Atom references by *car* tend to concentrate on a small number of atoms. The function *rplacd* increases static pointer locality, but *rplaca* is used idiosyncratically. Repeated reference to list cells is likely: over half of all references were to one of the ten most recently referenced cells. *Linearization* is the rearrangement of lists so that consecutive *cdr*'s are adjacent in memory whenever possible. This property deteriorates slowly after a list structure is linearized. If all of a program's lists are linearized, page faults are reduced slightly, but because of the high cost of a fault this small reduction has a large effect. [Abstract]

[Cohen 74]

J. Cohen and C. Zuckerman.

Two Languages for Estimating Program Efficiency.

Communications of the ACM 17(6):301-308, June, 1974.

Two languages enabling their users to estimate the efficiency of computer programs are presented. The program whose efficiency one wishes to estimate is written in the first language, a go-to-less programming language which includes most of the features of Algol 60. The second language consists of interactive commands enabling its users to provide additional information about the program written in the first language and to output results estimating its efficiency. Processors for the two languages are also described. The first processor is a syntax-directed translator which compiles a program into a symbolic formula representing the execution time for that program. The second processor is a set of procedures for algebraic manipulation which can be called by the user to operate on the formula produced by the first processor. Examples of the usage of the two languages are included. The limitations of the present system, its relation to Knuth's work on the analysis of algorithms, and some of the directions for further research are also discussed. [Abstract]

[Cohen 77]

J. Cohen and N. Carpenter.

A Language for Inquiring about the Run-time Behaviour of Programs.

Software--Practice and Experience 7:445-460, 1977.

This paper describes a language for studying the behaviour of programs, based upon the data collected while these programs are executed by a computer. Besides being a useful tool in debugging, the language is also valuable in the experimental evaluation of the complexity of algorithms, in studying the interdependence of conditionals in a program, and in determining the feasibility of transporting programs from one machine to another. The program one wishes to analyze is written in an Algol 60-like language; when the program is executed it automatically stores, in a data base, the information needed to answer general questions about computational events which occurred during execution. This information consists (basically) of the list of labels passed while the program is being executed, and the current values of the variables. Since the list of labels is describable by regular expressions, these expressions can also be used to identify specific subparts of the list and therefore allow access to the values of the variables. This constitutes the basis for the design of the inquiry language. The user's questions are automatically answered by a processor which inspects the previously generated data base. The paper also presents examples of the use of the language and describes the implementation of its processor. [Abstract]

[Comer 79]

Douglas Comer and Maurice H. Halstead.
A Simple Experiment in Top-Down Design.

IEEE Transactions on Software Engineering SE-5(2):105-109, March, 1979.

In this paper we: 1) discuss the need for quantitatively reproducible experiments in the study of top-down design; 2) propose the design and writing of tutorial papers as a suitably general and inexpensive vehicle; 3) suggest the software science parameters as appropriate metrics; 4) report two experiments validating the use of these metrics on outlines and prose; and 5) demonstrate that the experiments tended toward the same optimal modularity.

The last point appears to offer a quantitative approach to the estimation of the total length or volume (and the mental effort required to produce it) from an early stage of the top-down design process. If results of these experiments are validated elsewhere, then they will provide basic guidelines for the design process. [Abstract]

[Conti 79]

Dennis M. Conti.

Findings of the Standard Benchmark Library Study Group.

Technical Report 500-38, National Bureau of Standards Special Publication,
January, 1979.

This report presents the findings of a Government-industry study group investigating the technical feasibility of standard benchmark programs. As part of its investigation, the study group reviewed earlier efforts to develop and use standard benchmark programs. Several issues dealing with the implementation, maintenance, cost/benefit, and acceptability of standard benchmarks emerged as a result of this review. The problems encountered by the study group, notably the lack of an accepted definition of representativeness, prevented it from arriving at a definitive statement on feasibility. However, several areas were identified as topics requiring further investigation and are presented in this report. [Abstract]

[Courtois 77]

P. J. Courtois.

Decomposability: Queueing and Computer System Applications.

Academic Press, New York, 1977.

Large computing systems, like many complex systems, can be regarded as nearly completely decomposable systems -- systems in which the density of interactions between elements is low and the interconnection matrices are sparse. The book presents techniques for analysis of software and hardware systems that can be modeled as nearly-decomposable systems. [MS]

The theory of queueing networks, developed by Gordon, Jackson, and Newell in the late 1950s and early 1960s, lay almost unnoticed by computer analysts until, in 1971, Buzen discovered fast algorithms for calculating queue-length distributions in these networks. Since then interest in analytic models of computer performance has grown explosively. The theory has grown too: the models handle a variety of queueing disciplines, load dependent servers, multiple job classes, and service distributions of phase type. Computational algorithms have been developed for each of the extensions.

But the technology of queueing networks has reached its limits. The computational procedures do not handle priorities, blocking, or synchronized servers -- all of which occur in real computer systems. Moreover, the algorithms become unwieldy for systems of reasonable size, and they often exhibit numerical instabilities. The exact solution methods, which use brute force, are being dropped in favor of approximations, which use cunning.

P. J. Courtois has written a highly original monograph about the most powerful approximation method of which we know: decomposability. The method overcomes the difficulties of exact solutions to queueing problems.

The analysis of systems which are (nearly) decomposable has been attributed to Simon and Ando who, in 1961, reported on state-aggregation in linear models of economic systems. The key observation is that the matrices which describe complex systems tend to be mostly empty. Courtois's insight is that computer systems are hierarchies of components, within which interactions are strong and fast compared to interactions

between components at the same level. In developing the method of decomposability for analyzing computer systems, Courtois has also developed computational methods for any (nearly) decomposable system. [CR34612]

- [Crochow 69] J.M. Crochow.
Real-time graphic display of time-sharing system operating characteristics.
In *Proceedings AFIPS Fall Joint Computer Conference*, pages 374-386. AFIPS Press, 1969.
- [Crossman 79] Trevor D. Crossman.
Taking the Measure of Programmer Productivity.
Datamation:144-147, May, 1979.
- [Crowley 79] Charles Crowley and Gary Klimowicz.
A Note on Procedure Timing.
SIGPlan Notices 14(11), November, 1979.
The feasibility of procedure timing is discussed in relation to commonly used computers and operating systems. Some simple design criteria are described which facilitate such timing in an OS. Finally some observations are made on how clocks might be available at the hardware level to further facilitate procedure timing. [RTS]
- [Curtis 79] Bill Curtis, Sylvia B. Sheppard, Phil Millman, M. A. Borst and Tom Love.
Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics.
IEEE Transactions on Software Engineering SE-5(2):96-104, March, 1979.
Three software complexity measures (Halstead's E, McCabe's v(G), and the length as measured by number of statements) were compared to programmer performance on two software maintenance tasks. In an experiment on understanding, length and v(G) correlated with the percent of statements correctly recalled. In an experiment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with both the accuracy of the modification and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code. [Abstract]
- [Curtis 80] B. Curtis.
Measurement and experimentation in software engineering.
Special Issue of IEEE Proc., September, 1980.
This paper has two major parts. The first is a summary of metrics based on product properties useful to predict productivity and schedule. The second is a review of experimental evaluation of metrics proposed to capture programming attributes, mostly complexity. Close to one hundred papers are referenced. [LAB]
- [DACS 79] Data and Analysis Center for Software.
Quantitative Software Models.
Technical Report, U. S. Air Force, March, 1979.
This report has been prepared by the Data and Analysis Center for Software for the U. S. Air Force. The volume contains brief description of models and tabular classification of major contributions in the following three areas:

Life Cycle Cost/Productivity
Reliability/Error Analysis
Complexity

A fifty-two item bibliography is also attached. [LAB]

The purpose of this document is to disseminate information on the models and methods that encompass software life-cycle costs and productivity, software reliability and error analysis, and software complexity, and the data parameters associated with these models/methods. [Preface]

For each of 44 models, the report provides a summary of the purpose and characteristics of the model and a list of the model's parameters, outputs, etc. [MS]

[Dearnley 78]

P. Dearnley.

Monitoring database system performance.

The Computer Journal 21(1):15-19, 1978.

Monitoring is categorized into the areas of physical (sampling) and logical (modifying the source text). Logical monitoring is further broken down into the areas of cumulative (includes lower level routines) and differential. The DBMS was instrumented to record real and processor time on both a cumulative and a differential basis, in addition to frequency and number of transfers. [RTS]

[deFreitas 78]

S. L. de Freitas and P. J. Lavelle.

A Method for the Time Analysis of Programs.

IBM Systems Journal 17(1), 1978.

Discussed is a technique for investigating the efficiency of compiled programs. Based on research that uses FORTRAN as a test subject, the method is more widely applicable. Time analyses show programmers points at which efficiencies may be increased. Also discussed are uses of the technique for comparing the efficiencies of compilers and languages, and for making performance/cost analyses. Presented are validation data for the method under several sets of conditions. [Abstract]

[Denning 78]

Peter J. Denning.

The Operational Analysis of Queueing Network Models.

Computing Surveys 10(3):225-262, September, 1978.

Both Markovian queueing network theory and operational queueing network theory lead to the same mathematical equations. However, the derivations resulting from operational analysis are dependent upon one or more of the following operational principles: 1) All quantities are defined to be precisely measurable; 2) All assumptions are directly testable; 3) The system must be flow balanced; and 4) The system must be homogeneous.

These operational assumptions can be tested prior to applying the equations from this theory. Direct verification of the assumptions will allow analysts to more confidently apply queueing network analysis in their system evaluation endeavors. Hence, the difference between the Markovian and operational approaches is important.

The paper presents the results of operational queueing network theory and succinctly motivates its derivation. The concepts which are discussed include job flow balance, state transition balance, one-step behavior, homogeneity, and decomposition. Analysts not familiar with these concepts and their application should definitely study this tutorial. It is well written and highly recommended. [CR34957]

- [Doty 77] D. L. Doty, P. J. Nelson, and K. R. Stewart.
Software Cost Estimation Study; vol II: Guidelines for Improved Software Cost Estimating.
 Technical Report RADC-TR-77-220, Rome Air Development Center, August, 1977.
 This report contains guidelines for developing estimates of computer software cost. Consideration is first given to the initial program estimate which is often made with a paucity of supportive data. Adjustments are presented for modifying the estimate given the availability of additional data. Procedures are presented for assessing the affordability of the resulting estimates. Emphasis is placed on developing a conservative but reasonable best estimate for purposes of program budgeting. Separate consideration is given to steps that should be taken to bring the program in at or below budget. Frequently recurring problems are summarized in their time-phased order of occurrence. [Abstract]
- [Dunsmore 78] H. E. Dunsmore and J. D. Gannon.
 Programming Factors--Language Features that Help Explain Programming Complexity.
Proc. ACM 1978 Annual Conf. 2:554-560, 1978.
 Programming complexity (the amount of difficulty in constructing a program) may depend upon certain programming factors (choices of programming language features). Using program changes as a programming complexity measure, previous research has identified five potential programming factors. This paper suggests that subjects tend to use the same levels of these factors in two different programming languages, supporting the conjecture that these factors are elements of individual programming style. It also describes five potential programming factors, and although each of these has intuitive appeal, only average procedure length was related to programming complexity. [Abstract]
 The paper's abstract summarizes its contents well. This paper is interesting and well written; however, the reader should exercise caution in reaching conclusions based on the results presented. [CR34375]
- [Dunsmore 80] H. E. Dunsmore and J. D. Gannon.
 Analysis of the Effects of Programming Factors on Programming Effort.
The Journal of Systems and Software 1, 1980.
 Programming effort appears to be related to choices of programming language features which we call programming factors. A series of experiments was conducted investigating program construction, comprehension, and modification. Ease of construction seemed related to average nesting depth, percentage of global variables used for data communication, average variables referenced, and average live variables per statement. Data communication and live variables were shown to be related to ease of modification as well. [VRB]
- [Ellis 78] Clarence A. Ellis.
 Analysis of Some Abstract Measures of Protection in Computer Systems.
International Journal of Computing and Information Science 7(3):219-251, September, 1978.
 In this paper, Ellis has done a very good, very thorough, rigorous analysis of an uninteresting academic problem. It is embarrassing that such good work should be wasted on such an unrealistic approach to protection. The problem posed is based on protection by keys, similar to the method used in the IBM 360/370 computer storage protection system. The accesses of a set of subjects to a set of objects is to be controlled. A key is associated with each subject and each object. Access is allowed only if the key of the subject is properly related to the key of the object. A proper relation exists if the number of one-bits, resulting from a Boolean function of the keys of the subject and the object, exceeds a system threshold. By varying the function (equivalence, AND, ...) and the threshold, one can achieve various protection schemes.
 This is not an unrealistic basis for a protection scheme (IBM actually built one version of it), but Ellis then assumes 1) that objects are associated one-to-one with subjects, and

that a subject can only access its object; 2) keys are statically assigned; and 3) there are more subject/object pairs than there are distinct keys. In this situation, protection is not possible, so Ellis investigates how to minimize the amount of protection failure which can occur (worst case, average, ...), totally disregarding the fact that no one would ever build a system which is guaranteed to allow protection violations.

Given also that systems exist with the basic structure proposed, but with dynamic key assignments, we can only assume that since real systems are too difficult to analyze, a totally unreal system was analyzed instead. [CR33988]

[Elshoff 76]

James Elshoff.

An Analysis of Some Commercial PL/I Programs.

Transactions on Software Engineering SE-5(2):113-120, June, 1976.

The source code for 120 production PL/I programs from GM's commercial computing installations has been analyzed. Programs are considered with respect to five attributes: program size, program readability, program complexity, programmer discipline, and language use. [RTS]

The source code for 120 production PL/I programs from several General Motor's commercial computing installations has been collected. The programs have been scanned both manually and automatically. Some data from the scanning process are presented and interpreted.

The programs are considered with respect to five attributes: 1) the size of the programs, 2) the readability of the programs, 3) the complexity of the programs, 4) the discipline followed by the programmers, and 5) the use of the programming language. Each area is reviewed with pertinent data presented whenever it is available.

This report should be of interest to anyone involved with programming. The report helps explicitly identify some areas of programming in which a better job could be done. Although the programs analyzed are written in PL/I, those persons from installations using other languages, particularly Cobol, have indicated that the information presented is typical. [VRB]

[Ferrari 78]

Domenico Ferrari.

Performance of Computer Installations: Evaluation and Management.

In Domenico Ferrari, editor, *Proceedings of the International Conference on the Performance of Computer Installations*, pages 351. Elsevier/North-Holland, Inc., New York, June, 1978.

All computer installations are confronted with a variety of performance problems. When a new installation is to be set up, or an existing one is to be upgraded, such performance issues as the analysis of performance requirements for the system to be procured, the design of benchmarks or synthetic jobs to model the installation's projected workload, and the sizing of the system present themselves naturally to the parties involved. When the system is installed, it must be tuned to the actual workload, otherwise its performance would generally be lower than the one its capacity could produce. Since a system's performance is very sensitive to workload variations, and since the workload in most installations changes in time with many different patterns and frequencies, periodic retuning is often necessary or desirable. The procurement of systems software and applications software, that of external computing services, and various issues regarding machine room organization are some of the additional problems that are to be faced by scientific, administrative and commercial installations in their daily operation. To be successfully solved, these problems require that system performance be evaluated and managed. Evaluation methodologies, techniques and tools provide the knowledge on which performance managers are to base their decisions.

What is the state of the art in the areas of performance evaluation and performance management in 1978? Has significant progress been made toward the solution of such classical problems as workload characterization, benchmark design, capacity planning?? Is the field of performance evaluation ready to accept the challenges coming from new, rapidly emerging hardware and software techniques (distributed systems, database systems, multilevel automatically managed storage hierarchies, and so on)??

The International Conference on the Performance of Computer Installations (ICPCI 78), of which this volume contains the proceedings, was organized to provide an answer to these questions. [CR34345]

[Fitch 77]

John Fitch.

Profiling a Large Program.

Software--Practice and Experience 7:511-518, 1977.

A profiling technique (provided by the BCPL compiler) was used on a large algebra system, CAMAL, in order to determine where time was being used. Savings of up to 70 per cent as a result of small changes were reported. [RTS]

[Fitsos 79]

George P. Fitsos.

Software Science Counting Rules and Tuning Methodology.

Technical Report TR 03.075, IBM, September, 1979.

Programming Development at IBM's Santa Teresa Laboratory has been investigating "The Elements of Software Science" as defined by Dr. Maurice H. Halstead. A set of IMS and VSAM modules have been counted and were used to tune and specifically define the counting rules for assembler and PL/S. The method used for tuning the rules is presented, as are the rules themselves. Also discussed are some observations made during the tuning process. While the specific experiments were limited to assembler and PL/S languages, the methodology for tuning of rules would seem applicable to any language. [Abstract]

[Fitsos 80]

George P. Fitsos.

Vocabulary Effects in Software Science.

Technical Report TR-3926, IBM, January, 1980.

Programming Development at IBM's Santa Teresa Laboratory has been investigating the elements of software science as defined by Dr. Maurice H. Halstead of Purdue University. In conducting our experiments certain phenomena were observed, two of which are the subject of this report. The first relates to the number of unique operators. The number appears to be constant for a given higher level language. The second relates to program length. While the specific experiments were limited to assembler and PL/S languages, the observations would seem applicable to other programming languages. [Abstract]

[Fitzsimmons 78] Ann Fitzsimmons and Tom Love.

A Review and Evaluation of Software Science.

Computing Reviews 10(1):3-18, March, 1978.

During recent years, there have been many attempts to define and measure the "complexity" of a computer program. Maurice Halstead has developed a theory that gives objective measures of software complexity. Various studies and experiments have shown that the theory's predictions of the number of bugs in programs and of the time required to implement a program are amazingly accurate. It is a promising theory worthy of much more probing scientific investigation.

This paper reviews the theory, called "software science", and the evidence supporting it. A brief description of a related theory, called *software physics*, is included. [Abstract]

[Foxley 78]

E. Foxley and D.J. Morgan.

Monitoring the Run-Time Activity of Algol 68-R Programs.

Software--Practice and Experience 8:29-34, 1978.

A program profiling system for the Algol 68-R language is described. The system is a preprocessor using the language syntax analyzer. The profile consists of counts associated with each statement (some constructs were not monitored due to the lack of sophistication of the parser). [RTS]

- [Freburger 79] Karl Freburger and Victor R. Basili.
The Software Engineering Laboratory: Relationship Equations.
 Technical Report TR-764, University of Maryland, Computer Science Center,
 College Park, Maryland, May, 1979.
 Despite the fact that software costs are becoming a greater portion of the cost of using a computer system, very little research has been done to determine which factors impact the software development process. Presented are results of research into methods of estimating programming project variables such as effort, project duration, staff size and productivity. The results obtained are compared to the results of a previous study by Walston and Felix. [VRB]
- [Fries 76] M. J. Fries.
Software Error Data Acquisition.
 Technical Report RADC-TR-77-130, Rome Air Development Center, November, 1976.
 Software error data was collected from a large DOD system development project. The errors were analyzed and put into a predefined set of categories. As part of the effort, the times to find and fix the errors were calculated, and the phase of the development project in which the errors arose was determined. Study results were also compared to results of a similar type of study performed by a second contractor who performed analysis of data from another DOD software project. [Abstract]
- [Gaines 69] R. Gaines.
The Debugging of Computer Programs.
 PhD thesis, Princeton University, 1969.
- [Gannon 75a] J. D. Gannon and J. J. Horning.
 Language Design for Programming Reliability.
IEEE Transactions on Software Engineering 1(2), June, 1975.
- [Gannon 75b] John D. Gannon and James J. Horning.
 The Impact of Language Design on the Production of Reliable Software.
Proceedings of International Conference on Reliable Software:10-22, 1975.
 Gannon has performed a major experiment to measure the effect of nine specific language design decisions in one context. Analysis of the frequency and persistence of errors shows that several decisions had a significant impact on reliability. [Abstract]
- [Gannon 77] J. D. Gannon.
 An Experimental Evaluation of Data Type Conventions.
Communications of the ACM 20(8), August, 1977.
 This paper discusses an experiment that compares the programming reliability of subjects using a statically typed language and a "typeless" language. Two languages were designed by the author and attempts were made to make the two languages identical in all features not affected by the issue of data types. This particular experiment showed that in that particular environment, the features of a statically typed language increased programming reliability more than the features of a "typeless" language. [JES]

[Gehring 77]

Philip F. Gehring, Jr. and Udo W. Pooch.

A Quantitative Analysis of the Accuracy of Estimating in Software Development.

In *Proc. 16th Annual Technical Symposium on Systems Software: Operational Reliability and Performance Assurance*, pages 61-70. National Bureau of Standards, Gaithersburg Md., 1977.

The authors hypothesize that there are specific activities within a software development project whose estimate accuracy consistently reflects the estimate accuracy of the project as a whole. That is, if the resource consumption of these key activities is predicted accurately, then the resource consumption of the whole project will also be predicted accurately. Similarly, if a bad estimating job is done on the key activities, the estimate for the whole project will also be bad.

The authors tested this hypothesis by applying a statistical technique known as SEQUIN (for sequential item analysis) to data on some 39 software projects collected by the PARMIS project control system of the US Air Force Data System Design Center in Montgomery, Alabama. SEQUIN has previously been used to identify key questions in the Scholastic Achievement Test (SAT), such that scores on the key questions are strongly correlated to overall score. Such information is of use to educators in reducing the length of the SAT (not to mention its value to the students who take it).

The authors found that there are indeed specific key activities in the sense described above. The three activities leading the list are : 1) defining input data, 2) system design review, and 3) flowcharting of system processes.

The benefits of this knowledge are not made clear. The implication is that if effort were concentrated on producing an accurate estimate of, say, the resources to do input data definition, then overall project estimate accuracy could be significantly improved. But there is nothing in the remainder of the paper to suggest how the accuracy of key activities are in fact "key". It is a bit like knowing what links will cause a chain to fail, but being unable to do anything about those links.

The authors conclude that "estimation of software development continues to be as bad as, or worse than, it has ever been." They recommend a number of remedial measures, chief of which is replacing the classical phase structure of software development (feasibility, requirements, system design, etc.) by the Research-Development-Production model, in which each of the three phases is estimated on the basis of the preceding phase. "The RDP model will heighten management's awareness of the complexity of what has to be done to eliminate the historically impossible requirement to accurately estimate the cost and time for the entire software development project. [CR34785]"

[Gepner 78]

Herbert L. Gepner.

User Ratings of Software Packages.

Datamation 24(13):163-227, December, 1978.

Users of proprietary software packages rated 260 packages subjectively in the categories Overall Satisfaction, Throughput and Efficiency, Ease of Installation, Ease of Use, Documentation, Vendor Technical Support, and Training. Average opinion, rise, and brief notes on advantages and disadvantages are given for each. This article is abstracted from a Datapro 70 report. [MS]

[Gilb 77a]

T. Gilb.

Software Metrics Technology: Some Unconventional Approaches to Reliable Software.

In *Software Reliability*, pages 101-115. Infotech International Ltd., 1977.

A brief survey of techniques classified under the umbrella of "software metrics," i.e., they all relate to measurability of software reliability and closely related concepts, is presented. The subjects are classified into four categories, although most techniques could be said to belong to more than one:

1. Quantification of reliability

2. Automation of the reliability/maintainability task. This includes, for

example, assertion language tools, automatic program structuring, and automatic test case construction.

3. Management technology for reliable software. This includes, for example, design and code inspection, and the multi-element component comparison and analysis method.

4. Redundancy based reliability technology. This includes, for example, dual programming; advocated as the cheapest way to achieve reliability

The paper might be useful for project managers, although it is extremely superficial and fairly ad hoc. The author also seems to have a somewhat strange understanding of the notion "structured programming". [CR33990]

[Gilb 77b]

T. Gilb.

Distinct Software: A Redundancy Technology for Reliable Software.

In *Software reliability*, pages 117-133. Infotech International Ltd., Maidenhead, Berkshire, UK, 1977.

Distinctness is used to mean that there is some degree of redundancy in the software.

Some advantages of creating software distinctness for the software development process are described. 100% distinctness, i.e., there is for each program at least one other program which functionally performs identically, is argued to have great advantages during operational testing for automatic correction of bugs, maintenance, and adds far less to the total costs of the software than generally believed. Some advantages of data redundancy are also mentioned.

Finally, a number of mostly very favorable experiences with the application of distinct software are described.

The paper makes a reasonably good case for software distinctness as a means of achieving high reliability. The examples and experiences described are interesting and illustrative. [CR33968]

[Gilb 77c]

Tom Gilb.

Software Metrics.

Winthrop, 1977.

Gilb attempts to present a coherent overview of metric techniques and some practical applications.

[Goel 78]

Amrit L. Goel and K. Okamoto.

Bayesian Software Prediction Models.

Technical Report RADC-TR-78-155, Rome Air Development Center, July, 1978.

Five volumes.

These reports explore the use of a stochastic model for software failure phenomena for the case when the errors are not corrected with certainty. [MS]

[Gordon 79a]

Ronald D. Gordon.

Measuring Improvements in Program Clarity.

IEEE Transactions on Software Engineering SE-5(2):79-90, March, 1979.

The sharply rising cost incurred during the production of quality software has brought with it the need for the development of new techniques of software measurement. In particular, the ability to objectively assess the clarity of a program is essential in order to rationally develop useful engineering guidelines for efficient software production and language development.

A functional relation between the clarity of a program and the number and frequency of operators and operands which occur in the program is presented. This measure of program clarity provides an estimate of the amount of mental effort required to understand the program, assuming that the reader is fluent in the programming language employed.

This measure is tested by applying it to several published examples which demonstrate

improvements in program clarity. The objective assessment which is provided using this measure is found to agree with the experimental data gathered. [Abstract]

[Gordon 79b]

Ronald D. Gordon.

A Qualitative Justification for a Measure of Program Clarity.

IEEE Transactions on Software Engineering SE-5(2):121-128, March, 1979.

Several measures of program clarity have been proposed which attempt to assess the clarity of a program as a function of easily measured properties of the code. Such measures include the number of variables or statements, or the density of go to's.

The measure of program clarity, developed in the field of software science, equates the amount of mental effort required to understand a program with the ratio of program volume to implementation level. To be effective, a measure such as this should reflect the improvement in clarity which occurs when program transformations which make software easier to understand are applied.

The removal of each of six impurity classes from poorly written programs is studied. For a wide class of programs, purification reduces the amount of effort required for comprehension as predicted by the measure. [Abstract]

[Griswold 75]

Ralph Griswold.

A Portable Diagnostic Facility for SNOBOL4.

Software--Practice and Experience 5:93-104, 1975.

In programming systems based on abstract machine modeling concepts, the underlying structure of the abstract machine can be made available to the software implemented on it. The result is an unusual facility for diagnosis and exploration of software structure. Such a facility has been added to the macro implementation of SNOBOL4. This paper describes the nature of the facility, illustrates its use, and presents some results of using it for language implementation and development. [Abstract]

[Gupta 78]

Ram K. Gupta and Mark A. Franklin.

Working Set and Page Fault Frequency Paging Algorithms: a Performance Comparison.

IEEE Transactions on Computers 27(8):706-712, August, 1978.

The authors analyze the performance of the two paging algorithms mentioned in the title. Both analytic and empirical results are presented. The former are based on a program behavior model which assumes an inter-page-fault interval distribution which depends only on the size of the program's resident set. The empirical results are based on traces of two real programs. Performance comparison is based on the value of the space-time product, with time defined as the sum of virtual and page transfer times. Unfortunately, the value of the latter quantity is not given. The authors conclude that the PFF algorithm is more sensitive to the value of its control parameter than is the WS algorithm, and they therefore express preference for the latter. Their diagrams, however, show PFF to be clearly superior on one of the real program and, with proper tuning, capable of beating WS on two of the three analytic cases; thus the superiority of WS is far from proven, at least by the results of this paper. [CR34395]

[Halstead 77]

Maurice H. Halstead.

Operating and Programming Systems. Volume : Elements of Software Science.

Elsevier Computer Science Library, 1977.

This book contains the first systematic summarization of a branch of experimental and theoretical science dealing with the human preparation of computer programs and other types of written material. Application of the classical methods of the natural sciences demonstrates that even such relatively intangible objects as written abstracts and computer programs are governed by natural laws, both in their preparation and in their ultimate form. [VRB]

[Hansen 78]

Wilfred J. Hansen, Richard Doring, and Lawrence R. Whitlock.

Why an Examination was Slower Online than On Paper.

Int. J. Man-Mach. Stud. 10(5):507-520, September, 1978.

The authors looked at the use of a program which carries out interactive examinations on the PLATO system. They found that the length of time students took to complete interactive examinations were sometimes as much as twice the length of time taken to complete similar examinations given conventionally on paper. The authors sought to explain their finding by analyzing the two examination situations, including the collection of a limited amount of videotape evidence of students being examined. On the basis of this analysis, the authors propose suggestions, e.g., for faster display speeds which "may eventually" make PLATO a faster examination medium than paper. [CR34479]

[Hanson 78]

David Hanson.

Event Associations in SNOBOL4 for Program Debugging.

Software--Practice and Experience 8:115-129, 1978.

An event association facility for the SNOBOL4 programming language is described. This facility permits the execution of a programmer-defined function to be associated with the occurrence of a specified event. The set of valid events includes variable referencing, statement execution, program interruption, function call and return, and execution-time errors. The use and implementation of this facility is described. [RTS]

[Herd 77]

James H. Herd, John N. Postak, William E. Russell, and Kenneth R. Stewart.

Software Cost Estimation Study; vol I: Study Results.

Technical Report RADC-TR-77-220, Rome Air Development Center, June, 1977.

The study identified factors that have an adverse effect on software cost estimates, determined their impact on software cost estimates, discussed methods for controlling the effect of these factors, and developed an overall methodology for estimating the costs of software development. In addition to a generalized model for estimating software development costs, separate models have been generated for estimating the development cost of command and control, scientific, utility, and business software. [Abstract]

[Hoare 76]

C. A. R. Hoare.

The High Cost of Programming Languages.

In Software Systems Engineering, pages 413-429. Online Conferences Ltd.,

Uxbridge, UK, 1976.

This paper investigates the high cost of computer programming, including direct, indirect and consequential costs. It identifies eight major headings: organization, design, program construction, error, change, running costs, software procurement, and finally, delay. It then lists sixteen ways in which a programming language and its implementation can contribute to these costs, namely, by unfamiliarity, application-orientation, instability, independent compilation, debugging and optimizing compilers, machine dependence, obscurity of specification, freedom of expression, unreadability, inefficient translation, inefficient object code, unreliability, insecurity, lack of structure, illogicality, and complexity. The paper concludes with some advice of immediate and long-term benefit to programmers and their managers. [Abstract]

Since the paper is short in respect to the long list of topics, the discussion touches only the surfact of the problem of high cost of programming languages. But it is a good introduction and overview. [CR34245]

[Howden 78a]

William E. Howden.

*Theoretical and Empirical Studies of Program Testing.**Proc. 3rd International Conference Software Engineering* 20(4), April, 1978.

This paper starts off by distinguishing between the theoretical and empirical approaches to the study of program testing. Various theoretical (nonmathematical, graph-theoretical, and algebraic) and empirical approaches (path, branch, structured, special values, and symbolic testing, interface consistency, anomaly analysis, and specifications requirements) are described.

Next, the results of a research project are reported. A total of 28 errors were introduced into six working programs. Various test techniques were then used in attempts to identify these errors. Path testing was the most effective single technique. It found 18 of the errors. All techniques combined found 26 of the errors.

Two of the six programs were written in ALGOL. The others were written in COBOL, PL/I, FORTRAN, and PL360. Very little information is presented on the nature or size of these programs or on the nature of the errors introduced.

The author concludes that further empirical studies of testing are needed. This reviewer agrees. Considering the magnitude of the resources devoted to program testing, any guidance in the selection of the most effective test techniques for a given program (considering its nature, size, and source language) would be extremely valuable. [CR34958]

[Howden 78b]

William E. Howden.

*Functional Program Testing.**Technical Report DM-146-IR, University of Victoria, August, 1978.*

An approach to functional testing is described in which the design of a program is viewed as an integrated collection of functions. The selection of test data depends on the functions used in the design and on the value spaces over which the functions are defined. The basic ideas in the method were developed during the study of a collection of scientific programs containing errors. The method was the most reliable testing technique for discovering the errors. It was found to be significantly more reliable than structured testing. The two techniques are compared and their relative advantages and limitations are discussed. [Abstract]

[Howden 79]

William E. Howden.

*An Analysis of Software Validation Techniques for Scientific Programs.**Technical Report DM-171-IR, University of Victoria, March, 1979.*

Different empirical methods for assessing the effectiveness of software validation methods are discussed. *Error analysis* involves the examination of a collection of programs whose errors are known in advance. Each error is analyzed and the validation techniques are identified whose use would result in the discovery of the error. The results of an error analysis study of a package of Fortran scientific subroutines are described. The errors that were present in version five of the package and then later corrected in version six were analyzed. The results of the study indicate that the use of an integrated collection of static and dynamic analysis methods would have resulted in the discovery of the errors in edition five before its release. The paper is organized so that it describes the features of an integrated approach to validation as well as the effectiveness of individual methods. [Abstract]

[Huang 79]

J. C. Huang.

*Detection of Data Flow Anomaly Through Program Instrumentation.**IEEE Transactions on Software Engineering* SE-5(3):226-236 May, 1979.

A data flow anomaly in a program is an indication that a programming error might have been committed. This paper describes a method for detecting such an anomaly by means of program instrumentation. The method is conceptually simple, easy to use, easy to implement on a computer, and can be applied in conjunction with a conventional program test to achieve increased error-detection capability. [Abstract]

[Ingalls 71]

Daniel Ingalls.

Fete: A Fortran Execution Time Estimator.

Technical Report, Stanford University, February, 1971.

This report describes a preprocessor which takes a Fortran program and inserts code to accumulate counts for each statement. A postprocessor is also described which takes the modified program and correlates the text with the final counter values. Although counts rather than times are stored, the postprocessor tries to estimate the cost of each statement as a function of the operators which compose that statement. [RTS]

[Jackson 78]

Richard H.F. Jackson and John M. Mulvey.

A Critical Review of Comparisons of Mathematical Programming Algorithms and Software.

J. Res. N.B.S. 83(6):563-585, Nov.-Dec., 1978.

This paper surveys fifty articles spanning the period 1953-77 which report the computational testing of mathematical programming algorithms. The authors' intention is to document the methods employed in conducting these experiments, including the selection of problems, algorithm description, experiment design, and the form of reported results.

The survey is arranged according to the following topics: elements of the experiment (algorithms, software, problem class); experiment design (test problems, computer environments, experiment controls); empirical results (performance measures, statistical methods, mathematical checks, reporting of empirical evidence, interpretation of results); and suggestions for future work.

The papers included for critical review cover the major areas of mathematical programming: LP, IP, unconstrained optimization, shortest paths, NLP, networks (mincost flow), geometric problems, system of nonlinear equations, quadratic programs, and knapsack problems.

It may be useful to give some examples from the survey which the reviewer finds quite typical of many papers reporting computational experiments with mathematical software.

- Insufficient description of the algorithms.
- Lack of attention to the software elements: portability, availability, ease of use, and tolerance setting.
- Lack of information related to the methods of generating test problems.
- Preprocessing of problems (scaling, data sorting, etc.) not indicated in many papers.
- Computer environment and experiment controls (e.g., computer, compiler, operating system) not always described in sufficient detail. In fact, several papers do not name the computer used.
- Different measures of performance used by researchers: CPU time, iterations, function evaluations, etc. Storage requirements not always defined.
- Limited use of statistical methods to analyze computational experiments.
- Lack of concise statements indicating the purpose of the computational experiment and the limitations of the study.

The authors of the paper suggest that fundamental research in the area of computer-algorithm performance is long overdue. Some initial attempts to rectify the situation in this area have been made. One of the authors of the reviewed paper has co-authored a report addressing the issue under discussion. [CR34894]

[Jeffery ??]

D. R. Jeffery and M. J. Lawrence.

An Inter-Organizational Comparison of Programming Productivity.

Technical Report, University of New South Wales, Department of Information Systems, ??.

The factors which influence program size and program development time have been investigated across three dissimilar organizations. Data on a total of 93 COBOL programs has been collected and analyzed. Eighteen variables covering the characteristics of the program, programmer and programming environment were recorded. Program size and program development time were found to have a strong program characteristic and organization dependency. Programmer characteristics did not appear to play a role in influencing program size or program development time. The best determinant of program development time was found to be procedure division lines of code, which gave a simple regression R^2 in excess of .79 for the two organizations using well formulated programming standards. Productivity measures based on lines of code per hour are shown to be misleading in inter-organizational comparisons. [VRB]

[Johnston 70]

T.Y. Johnston and R.H. Johnston.

Program Performance Measurement.

SLAC User Note 33, Revision 1, Stanford Linear Accelerator Center, 1970. Stanford, CA.

[Jones 78]

T. C. Jones.

*MEasuring Programming Quality and Productivity.**IBM Systems Journal* 17(1), 1978.

Discussed is the unit-of-measure situation in programming. An analysis of common units of measure for assessing program quality and programmer productivity reveals that some standard measures are intrinsically paradoxical. Lines of code per programmer-month and cost per defect are in this category. Presented here are attempts to go beyond such paradoxical units as there. Also discussed is the usefulness of separating quality measurements into measures of defect removal efficiency and defect prevention, and the usefulness of separating productivity measurements into work units and cost units. [Abstract]

[Kleinrock 75a]

L. Kleinrock.

Queuing Systems, Volume I: Theory.

John Wiley & Sons, 1975.

[Kleinrock 75b]

L. Kleinrock.

Queuing Systems, Volume II: Applications.

John Wiley & Sons, 1975.

[Knuth 71]

Donald Knuth.

*An Empirical Study of FORTRAN Programs.**Software--Practice and Experience* 1:105-133, 1971.

Static and dynamic statistics on a sample of programs were gathered. The principle conclusion is the importance of a program profile, which is a table of frequency counts which record how often each statement is performed in a typical run. It appears that the n th most important statement of a program from the point of execution time accounts for about $(a-1)a^{-n}$ of the running time, for some a and for small n (very approximately). Generally less than 4 per cent of a program accounts for more than half of its running time. [RTS]

A sample of Fortran programs was analyzed to discover what programmers "really" do. Analysis techniques included static counts of syntactic constructs, dynamic counts of actual executions, and detailed examination of inner loops. Statistical results and some of their apparent implications are presented. [MS]

- [Knuth 73] Donald Knuth and Francis Stevenson.
Optimal Measurement Points for Program Frequency Counts.
BIT 13:313-322, 1973.
A procedure recently devised by A. Nahapetian reduces an arbitrary flowchart to the minimal one, on which program frequencies can be measured. The algorithm is optimal, in that the minimum number of measurements is determined. An example implementation in Simula is given. [RTS]
- [Laemmel 78] A. Laemmel and M. Shooman.
Software Modeling Studies.
Technical Report RADC-TR-78-4, Rome Air Development Center, April, 1978.
This report discusses the application of concepts of statistical language theory (Zipf's Laws) to the derivation of formulas for measuring program and language complexity. Experimental data from several different programs and programming languages, such as PL/I, assembly and FORTRAN, is presented which is used to verify the necessary underlying assumption and to verify formulas for program length by comparison with actual statistics. Finally, the derived formulas are compared with those of Software Physics derived by Halstead. [Abstract]
- [Lehman 80] M. M. Lehman.
Programs, programming and the software life cycle.
Special Issue of IEEE Proc., September, 1980.
This paper clarifies the difference between the evolution dynamics of program development and the dynamics of program execution. Also, a classification of large programs is offered, following the difficulty and severity of continuous enhancement and maintenance. Extensive bibliography attached. [LAB]
- [Littlewood 75] B. Littlewood.
A Reliability Model for Markov Structured Software.
Proceedings of International Conference on Reliable Software:204-207, 1975.
A system is considered in which switching takes place between sub-systems according to a continuous parameter Markov chain. Failures may occur in Poisson processes in the sub-systems, and in the transitions between sub-systems. All failure processes are independent. The overall failure process is described exactly and asymptotically for highly reliable sub-systems. An application to process-control computer software is suggested. [Abstract]
- [Love 77] Tom Love.
An Experimental Investigation of the Effect of Program Structure on Program Understanding.
Proc. ACM Conference on Language Design for Reliable Software:105-113, March, 1977.
A within-subjects experimental design was used to test the effect of two variables on program understanding. The independent variables were complexity of control flow and paragraphing of the source code. Understanding was measured by having the subjects memorize the code for a fixed time and reconstruct the code verbatim. Also, some subjects were asked to describe the function of the program after completing their reconstruction. The two groups of subjects for the experiment were students from an introductory programming class and from a graduate class in programming languages.

The major findings were that paragraphing of the source had no effect for either group of subjects but that programs with simplified control flow were easier for the computer science students to understand as measured by their ability to reconstruct the programs. The dependent variable, rated accuracy of their description of the programs functions, did not differ as a function of either independent variable.

The paper is concluded with a description of the utility of this experimental approach relative to improving the reliability of software and a discussion of the importance of these findings. [CR34498]

[Lyness 79]

J. N. Lyness.

A Benchmark Experiment for Minimization Algorithms.

Math. Comput. 33(145):249-264, January, 1979.

Among the ground rules for the empirical evaluation of an algorithm one should list a) the need for a statistical approach which is not unduly influenced by an occasional lucky or unlucky break, b) the need for a parametric set of test problems so that the algorithm can be tested under all conditions, easy through impossible, c) the need for condensing a large volume of test results into a small number of figures which characterize the properties of the algorithm.

In test minimization programs the author's proposal for b) is a multiparameter family of "Helical Valley Objective Functions". He accommodates a) and c) by producing a probability distribution function for costs. This has the merit that occasional failures can be seen in proper perspective, without going to the extreme of either ignoring or treating them as fatal flaws. He demonstrates the test sequence on several standard programs. [CR34822]

[Lyon 75]

Gordon Lyon and Rona Stillman.

Simple Transforms for Instrumenting FORTRAN Decks.

Software--Practice and Experience 5:347-358, 1975.

A preprocessor is described which divides the source into code segments and adds calls to a monitoring routine which accumulates counts at the segment (statement) level. A division of monitoring is also given: clock interrupts via the operating system, counters inserted into a program, calls to a system clock, and event driven hardware probes. [RTS]

[Mamrak 79]

Sandra A. Mamrak and Paul D. Amer.

A Methodology for the Selection of Interactive Computer Services.

Technical Report 500-44, National Bureau of Standards Special Publication,
January, 1979.

This publication addresses the comparison and selection of remote access interactive computer services. The comparison methodology presented relies principally on the statistical analysis of measurement data obtained from the interaction between a computer service and a user. One of the most important properties of the methodology is that it incorporate confidence statements about the probability of having made a correct selection. Experimental data are presented to illustrate an application of the methodology, and serve as a basis for a discussion of the cost and appropriateness of using the methodology in various procurement efforts. [Abstract]

[Matwin 76]

S. Matwin and M. Missala.

A Simple, Machine Independent Tool for Obtaining Rough Measures of Pascal Programs.

SIGPlan Notices 11(8):42-45, August, 1976.

This paper describes a profiling system written in standard Pascal which consists of a preprocessor and a postprocessor. The preprocessor inserts statements into the source of the program to be monitored. As the program runs, it outputs an event record each time a routine starts or returns. A postprocessor uses this event file to determine execution counts and times for the routines in the program. [RTS]

[McCabe 76]

Thomas J. McCabe.

A Complexity Measure.

IEEE Transactions on Software Engineering SE-2(4), December, 1976.

This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran programs are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size

AD-A087 412

YALE UNIV. NEW HAVEN CT DEPT OF COMPUTER SCIENCE

F/G 9/2

DRAFT SOFTWARE METRICS PANELS FINAL REPORT. PAPERS PRESENTED AT--ETC(U)

JUN 80 A J PERLIS, F G SAYWARD, M SHAW

N00014-79-C-0672

UNCLASSIFIED

RR-182/80

NL

4 4

AL



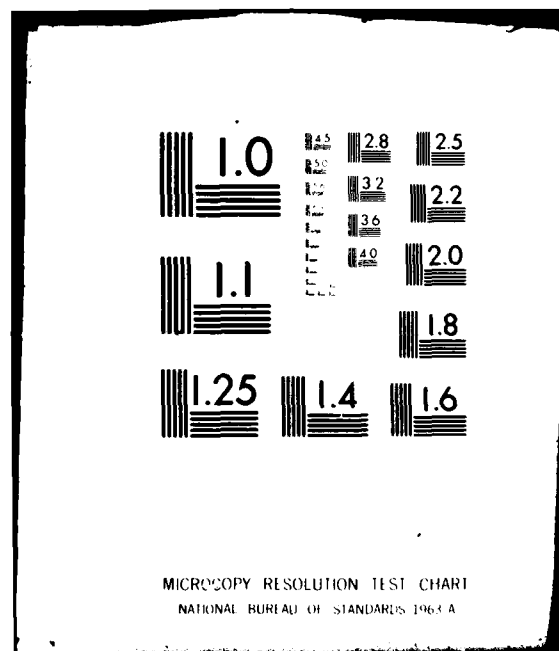
END

DATE

TIMED

9-80

OTIC



(adding or subtracting functional statements leaves complexity unchanged) and complexity depends only on the decision structure of a program.

The issue of using nonstructured control flow is also discussed. A characterization of nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.

The last section of this paper deals with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced. [VRB]

[McKissick 79]

John McKissick, Jr. and Robert A. Price.

Software Quality Assurance.

Proceedings 1979 Annual Reliability and Maintainability Symposium, 1979.

The continuing need for improved computer software demands improved software development techniques such as the Software Development Notebook. The organization, content, use and audit of Software Development Notebooks are documented in this paper. Experience and results from the application of this technique are also presented.

[Abstract]

[Millbrant 74]

W.W. Millbrant and J. Rodriguez-Rosell.

An interactive software engineering tool for memory management and user program evaluation.

In *Proceeding AFIPS NCCE*, AFIPS Press, 1974.

[Mills 76]

Harlan D. Mills.

Software Development.

IEEE Transactions on Software Engineering SE-2(4), December, 1976.

Software development has emerged as a critical bottleneck in the human use of automatic data processing. Beginning with ad hoc heuristic methods of design and implementation of software systems, problems of software systems, problems of software maintenance and changes have become unexpectedly large. It is contended that improvement is possible only with more rigor in software design and development methodology. Rigorous software design should survive its implementation and be the basis for further evolution. Software development should be done incrementally, in stages with continuous user participation and replanning, and with design-to-cost programming within each stage. [VRB]

[Model 78]

M. Model.

Monitoring System Behavior in a Complex Computational Environment.

PhD thesis, Stanford University, January, 1978.

Stan-CS-79-701.

This work is directed at the development of appropriate monitoring tools for complex systems, in particular, the representation systems of Artificial Intelligence research. The first half of this work provides the foundation of the design approach put forth and demonstrated in the second. Certain facts concerning limitations on human information processing abilities which formed the background for much of the research are introduced. Observation of program behavior ('monitoring') is shown to be the main function of most debugging tools and techniques.

The second half presents an approach to the design of monitoring facilities for complex systems. A new concept called 'meta-monitoring' replaces traditional dumps and traces with selective reporting of high-level information about computations. The importance of the visually-oriented analogical presentation of high-level information and the need to take into account differences between static and active processes are stressed. A generalized method for generating descriptions of system activity is developed. Some specific display-based monitoring tools and techniques which were implemented for this work are exhibited. [Abstract]

[Mohanty 79]

Siba N. Mohanty.

Models and Measurements for Quality Assessment of Software.

Computing Surveys 11(3), September, 1979.

Several software quality assessment methods which span the software life cycle are discussed. The quality of a system design can be estimated by measuring the system entropy function or the system work function. The quality improvement due to reconfiguration can be determined by calculating system entropy loading measures. Software science and Zipf's law are shown to be useful for estimating program length and implementation time. Deterministic and statistical methods are presented for predicting the number of errors. Testing theory is useful in planning the program test process; as discussed in this paper, it includes measurement of program structural characteristics to determine test effectiveness and test planning. Statistical models for estimating software reliability are also discussed. [VRB]

[Motley 77]

R. W. Motley and W. D. Brooks.

Statistical Prediction of Programming Errors.

Technical Report RADC-TR-77-175, Rome Air Development Center, May, 1977.

This report presents and discusses the results obtained for statistical predictions of programming errors using multiple linear regression analysis. Programming errors were predicted from linear combinations of program characteristics and programmer variables. Each of the program characteristic variables were considered to be measures of the program's complexity and structure. Two distinct data samples comprising 783 programs with approximately 297,000 source instructions written for command and control software applications were analyzed. Background data on both samples is provided which includes discussions related to each sample's software development environment, testing conditions, predictor variables, definition of programming errors, and general data characteristics. Results are presented which give the prediction equations obtained and a discussion of the predictability of errors and error rate in each sample. Conclusions of the study and recommendations for further research are also provided. [Abstract]

[Musa 75]

John D. Musa.

A Theory of Software Reliability and Its Application.

IEEE Transactions on Software Engineering SE-1(3), September, 1975.

An approach to a theory of software reliability based on execution time is derived. This approach provides a model that is simple, intuitively appealing, and immediately useful. The theory permits the estimation, in advance of a project, of the amount of testing in terms of execution time required to achieve a specified reliability goal (stated as a mean time to failure (MTTF)). Execution time can then be related to calendar time, permitting a schedule to be developed. The reliability model that has been developed can be used in making system tradeoffs involving software or software and hardware components. The model has been applied to four medium-sized software development projects, all of which have completed their life cycles. [VRB]

[Myers 78a]

Ware Myers.

A Statistical Approach to Scheduling Software Development.

Computer 11(12):23-38, December, 1978.

This paper is mostly a summary of Putnam's work in resource estimation on large-scale software developments. In fact, a more appropriate title would have been "The Putnam/Norden Statistical Approach to Scheduling Software Development" since it is essentially about Putnam's extensions to Norden's original work at IBM in the early 1960s.

It is assumed that there is a natural cost curve governing software development. This curve is stated to be a Rayleigh curve, $y = 2KaI \exp(-aI^2)$, where y is the expenditure on a project per year, I is the elapsed time in years, K is the total project cost (including maintenance), and a is a shape parameter of the curve. With data obtained from the US Army Computer Systems Command, Putnam has shown that real projects agree with this curve quite closely.

If td is defined to be the development time (or the time until the Rayleigh curve reaches a maximum), then the Rayleigh curve can be rewritten as a differential equation of the form $y = (K/td^2) \exp(-1/2) / [2td^2]$. (Note: The paper contains a typographical error at this point.) The term K/td^2 is defined to be the difficulty of a project, and again empirical data seems to agree with this assumption.

Comparing these equations with the gradient of a function, the effects of modifying time (t) or cost (y) on the difficulty can be measured, and a potential practical measure results. For example, if a project is required in 10 percent less time with the same staff, the increase in difficulty can be measured. Or if 10 percent less time is needed, how much larger a staff is needed to keep the same level of difficulty. The paper points out that there are limits to how much of a tradeoff can be made for time and people--the two are not totally interchangeable.

This paper is a well written summary of Putnam's work. Unfortunately, it suffers from the same deficiency as much of his other work--a lack of clear explanation of the underlying theory. To the uninitiated or skeptical the basic Rayleigh curve assumption looks quite ad hoc; it is, however, based upon a formal theory. The curve derives from hardware reliability theory, and is based upon the following assumptions:

1. A project has a fixed number of problems to solve.
2. Each problem requires so much time to be solved.
3. Solving a problem leads to fewer remaining unsolved problems (e.g., fixing a bug does not introduce a new bug).
4. Increasing the number of people leads to an increased rate of problem solving.

The Rayleigh curve is a natural consequence of these assumptions. While the assumptions may not be totally accurate, they are a good beginning to an explicit mathematical theory of cost estimation. Unfortunately, it took this reviewer several months to track down any clearly written reference to this development. A good theoretical basis is needed to increase acceptance of the empirical data, and I hope that more will be written about the underlying theory. (The same comments apply to the other phases of the theory as well.)

While the formalism may not be totally correct, this paper is quite important. This is one of the few areas of research that is trying to get a firm grip on software costs and estimates. Software is just too expensive and unreliability is too dangerous for such current estimates as "about 10 lines of code a day" to be satisfactory in estimating the costs of multi-million dollar projects. [CR34649]

[Myers 78b]

Glenford J. Myers.

A Controlled Experiment in Program Testing and Code Walk-Throughs/Inspections.

Communications of the ACM 21(9):760-768, September, 1978.

This very carefully detailed paper concerns the testing of a PL/I program patterned after the by now famous text reformatter program by P. Naur. Although the abstract states that seven methods were used to test the program, there were only three essentially different methods used; the test results of the application of these three were combined (ex post facto) in four additional ways. The three methods used are: 1) computer-based testing with specifications, but with the listing; 2) computer-based testing with both listing and specifications; 3) noncomputer-based testing by a team of three programmers using the walkthrough/inspection method on the listing. Groups were formed for each of the three methods and balanced as much as possible with respect to testing experience, knowledge of PL/I, and experience with walk-through procedures. [CR34125]

[Naur 78]

P. Naur.

Software Reliability,

In R. K. D. Rees, *Software Reliability*. Infotech International Ltd., Maidenhead, Berkshire, UK, 1978.

This excellent paper gives a very good clarification of the concept of software reliability. The author argues, with convincing examples, that reliability can only be understood as a relation between a certain system behavior and certain human expectations. In other words, we cannot judge the reliability of a system by observing its behavior. We have to compare this behavior with some expected standard of behavior. Reliability cannot be reduced to a single dimension, therefore any attempt to measure it along a one dimensional scale is misleading.

The author also briefly discusses reliability and correctness, reliability as related to systems controlled by software, stability of software systems, and reliable software design policies. [CR33969]

[Parr 80]

F. N. Parr.

An Alternative to the Rayleigh Curve Model for Software Development Effort.

IEEE Transactions on Software Engineering SE-6(3), May, 1980.

A new model of the software development process is presented and used to derive the form of the resource consumption curve of a project over its life cycle. The function obtained differs in detail from the Rayleigh curve previously used in fitting actual project data. The main advantage of the new model is that it relates the rate of progress which can be achieved in developing software to the structure of the system being developed. This leads to a more testable theory, and it also becomes possible to predict how the use of structured programming methods may alter patterns of life cycle resource consumption. [VRB]

[Perrott 77]

R.H. Perrott and A.K. Raja.

Quasiparallel Tracing.

Software--Practice and Experience 7:483-492, 1977.

A description of different methods of program tracing in a quasiparallel environment, as well as a specific automatic tracing facility, is given. This facility traces process activation, entrance, exit, waiting, signaling, resuming, and restarting of monitor processes. Space and time overhead is examined. [RTS]

[Putnam 78]

Lawrence H. Putnam.

A General Empirical Solution to the Macro Software Sizing and Estimating Problem.

IEEE Transactions on Software Engineering SE-4(4), July, 1978.

Application software development has been an area of organizational effort that has not been amenable to the normal managerial and cost controls. Instances of actual costs of several times the initial budgeted cost, and a time to initial operational capability sometimes twice as long as planned are more often the case than not.

A macromethodology to support management needs has now been developed that will produce accurate estimates of manpower, costs, and times to reach critical milestones of software projects. There are four parameters in the basic system and these are in terms managers are comfortable working with -- effort, development time, elapsed time, and a state-of-technology parameter.

The system provides managers sufficient information to assess the financial risk and investment value of a new software development project before it is undertaken and provides techniques to update estimates from the actual data stream once the project is underway. Using the technique developed in the paper, adequate analysis for decisions can be made in an hour or two using only a few quick reference tables and a scientific pocket calculator. [VRB]

[QSM 79]

**Workshop on Quantitative Software Models for Reliability, Complexity, and Cost:
An Assessment of the State of the Art.**

Workshop held at Concord Hotel, Kiamesha Lake, New York, in October, 1979,
IEEE Catalog No. TH0067-9.

This is the proceedings of a workshop on quantitative software models for reliability, complexity and cost, and contains a large number of papers related to both models and metrics of software development. Included are the evaluation of many models by various organizations. [VRB]

[Roberts 80]

Teresa Lynn Roberts.

Evaluation of Computer Text Editors.

PhD thesis, Stanford University, 1980.

[Robinson 79]

John T. Robinson.

Some Analysis Techniques for Asynchronous Multiprocessor Algorithms.

IEEE Transactions on Software Engineering SE-5(1):24-31, January, 1979.

Efficient algorithms for asynchronous multiprocessor systems must achieve a balance between low process communication and high adaptability to variations in speed. Algorithms that employ problem decomposition may be classified as static (in which decomposition takes place before execution) and dynamic (in which decomposition takes place during execution). Static and dynamic algorithms are particularly suited for low process communication and high adaptability, respectively. For static algorithms the following analysis techniques are presented: finding the probability distribution of execution time, deriving bounds on mean execution time using order statistics, finding asymptotic mean speedup, and using approximations. For dynamic algorithms the technique of modeling using a queueing system is presented. For each technique, and example application to parallel sorting is given. [Abstract]

[Rubey 75]

R. J. Rubey.

Quantitative Aspects of Software Validation.

Proceedings of International Conference on Reliable Software:246-251, 1975.

This paper discusses the need for quantitative descriptions of software errors and methods for gathering such data. The software development cycle is reviewed and the frequency of the errors that are detected during software development and independent validation are compared. Data obtained from validation efforts are presented, indicating the number of errors in 10 categories and three severity levels; the inferences that can be drawn from this data are discussed. Data describing the effectiveness of validation tools and techniques as a function of time are presented and discussed. The software validation cost is contrasted with the software development cost. The applications of better quantitative software error data are summarized. [Abstract]

[Russell 69]

E.C. Russell, Jr.

Automatic Program Analysis.

PhD thesis, UCLA, 1969.

Report 69-12.

[Ryder 79]

Barbara G. Ryder.

Constructing a Call Graph of a Program.

IEEE Transactions on Software Engineering SE-5(3):216-226, May, 1979.

The proliferation of large software systems written in high level programming languages insures the utility of analysis programs which examine interprocedural communications. Often these analysis programs need to reduce the dynamic relations between procedures to a static data representation. This paper presents one such representation, a directed, acyclic graph named the call graph of a program. We delineate the programs representable by an acyclic call graph and present an algorithm for constructing it using the property that its nodes may be linearly ordered. We prove the correctness of the algorithm and discuss the results obtained from an

implementation of the algorithm in the PFORT Verifier. [Abstract]

[Rye 77]

P. Rye, F. Bamberger, W. Ostanek, N. Brodeur, and J. Goode.

Software Systems Development: A CSDL Project History.

Technical Report RADC-TR-77-213, Rome Air Development Center, June, 1977.

This report provides a description of the data delivered to RADC for inclusion in a Software Data Repository. The data consists of a complete history of software modifications to the APOLLO on-board flight software for the period 1967 through 1971. Background material on the project that was the source of the data is provided, as well as tabular and graphic summaries of the data. Some recommendations for future work are made. [Abstract]

[Saltzer 70]

Jerome Saltzer and John Gintell.

The Instrumentation of MULTICS.

Communications of the ACM 13(8):495-500, August, 1970.

An array of measuring tools devised to aid in the implementation of a prototype computer utility is discussed. These tools include 1) a hardware calendar clock (52 bit, 1 microsecond resolution) and an associated match register; 2) a memory reference counter; 3) an input/output channel which can be used by an attached processor to read memory; 4) a general metering package which records time spent executing selectable supervisor modules while the system is running; 5) a segment utilization metering facility which periodically probes for the current segment number; 6) a facility which records on a per-segment basis the number of missing pages and segments encountered during execution in that segment; 7) a tool which counts the number of times procedures are called; 8) a software package implemented on a PDP-8 which utilizes the special I/O channel (3); 9) the CLI, which prints out the time of day, the CPU time, and the number of times the process had to wait for a page to be brought in after every 'ready message'; 10) a ring buffer containing the segment, page number, and time of day of the last 256 missing pages of the process under measurement; 11) a package to monitor the effect of the system's multiprogramming effort of an individual program; 12) a script driver implemented on a PDP-8; and 13) an internal script driver. [RTS]

[Sammet 70]

Jean E. Sammet.

Perspective on Methods of Improving Software Development.

Software Engineering 1, 1970.

The concept of improving software development is important but ambiguous. The two major difficulties lie in attempting to measure the various aspects and the need to recognize the vast amount of tradeoffs required. By using a formula which permits (and requires) the manager to assign weights to the various facets, some quantitative information can be obtained on various tradeoffs. The productivity of an individual is important, but is only one facet of the entire development cycle. A number of specific technical and management techniques for improving software development have been described. [Summary]

[Sammet 71]

J. E. Sammet.

Problems in, and a Pragmatic Approach to Programming Language Measurement.

In *Proceedings AFIPS Fall Joint Computer Conference*, pages 243-251. AFIPS, 1971.

This appears to be the first published paper to discuss measurements in programming languages (as contrasted with measurements in or of programs). The paper describes the problem and its importance. Part of the paper discusses the need for consideration of non-technical (as well as technical) issues in selecting a programming language. A weighted scoring technique is described and illustrated with one example. A second part of the paper discusses the problem of defining the terms "dialect" and "language-L-like" and presents a method for measuring numerically the amount of deviation of one language from another. [JES]

[Satterthwaite 72] E. Satterthwaite.

Debugging Tools for High Level Languages.

Software--Practice and Experience 2:197-217, 1972.

The design of an integrated programming and debugging system using the language Algol W is described. The debugging tools are based entirely upon the source language but can be efficiently implemented. The most novel such tool is a selective trace, automatically controlled by execution frequency counts. System performance information is included. [Abstract]

[Schafer 79]

R. E. Schafer, J. E. Angus, J. F. Alter, and S. E. Emoto.

Validation of Software Reliability Models.

Technical Report RADC-TR-79-147, Rome Air Development Center, June, 1979.

This report presents the results of a study and investigation of software reliability models. In particular, the purpose was to investigate the statistical properties of selected software reliability models, including the statistical properties of the parameter estimates, and to investigate the goodness of fit of the models to actual software error data. The results indicate that the models fit poorly, generally due to in most part the vagaries of the data rather than shortcomings of the models. [Abstract]

[Schneiderman 80]

B. Schneiderman.

Software Psychology: Human Factors in Computer and Information Systems.

Winthrop, Cambridge, Massachusetts, 1980.

This is a very new book which discusses a number of issues which are relevant to software metrics. A large number of experiments which have been conducted by many people over a period of time are described. A chapter discusses the software metrics which have been developed by various people. While the book contains several chapters which are not related to metrics, it nevertheless appears to be the most complete single source of material on the numerous approaches to software metrics and experiments. [JES]

[Sevcik 74]

Kenneth C. Sevcik.

Computer System Modelling and Analysis: Assessing Some Common Assumptions.

Proc. Seventh Hawaii International Conference on System Sciences:37-39, 1974.

Certain assumptions have been made frequently in studying analytical models of computer systems. Before applying conclusions derived from such models, the validity of the assumptions must be judged. Any assumption of questionable validity must be further investigated to determine extent to which its variation can affect the conclusions drawn from the model. Here, we investigate assumptions about various quantities of significance in several types of computer system models. Some suggestions for increasing the relevance of future modelling studies are given. [Abstract]

[Shaw 74]

Mary Shaw.

Reduction of Compilation Costs Through Language Contraction.

Communications of the ACM 16(5):245-250, May, 1974.

Simpler languages tend to have simpler compilers than more complex languages, but programs in simpler languages may have to be larger to accomplish the same tasks. This paper uses a combination of measurement and simulation techniques to establish the nature of the tradeoff between program size and language size. [MS]

[Shaw 79]

Mary Shaw.

A Formal System for Specifying and Verifying Program Performance.

Technical Report CMU-CS-79-129, Carnegie-Mellon University, June, 1979.

Formal techniques for specifying performance properties of programs (e.g., execution time) and for verifying the correctness of these specifications are developed. These techniques are extensions of well-known predicate transformer techniques for specifying purely functional properties of programs. [Abstract]

[Shaw 80]

Mary Shaw, Guy T. Almes, Joseph M. Newcomer, Brian K. Reid, and Wm. A. Wulf.

*A Comparison of Programming Languages for Software Engineering.**Software -- Practice and Experience*, 1980.

Four programming languages (Fortran, Cobol, Jovial and the proposed DOD standard) are compared in the light of modern ideas of good software engineering practice. The comparison begins by identifying a core for each language that captures the essential properties of the language and the intent of the language designers. These core languages then serve as a basis for the discussion of the language philosophies and the impact of the language on gross program organization and on the use of individual statements. [Abstract]

[Sheppard 79]

Sylvia B. Sheppard, Phil Milliman and Bill Curtis.

*Factors Affecting Programmer Performance in a Debugging Task.**General Electric Software Management Research TR-79-388100-5*, February, 1979.

This report is the third in a series investigating characteristics of software which are related to its psychological complexity. Three independent variables, length of program, complexity of control flow, and type of error, were evaluated for three different Fortran programs in a debugging task. Fifty-four experienced programmers were asked to locate a single bug in each of three programs. Documentation consisted of input files, correct output, and erroneous output. Performance was measured by the time to locate and successfully correct the bug.

Small but significant differences in time to locate the bug were related to differences among programs and presentation order. Although there was no main effect for type of bug, there was a large program by error interaction suggesting the existence of context effects. Among measures of software complexity, Halstead's E proved to be the best predictor of performance followed by McCabe's $V(G)$ and the number of lines of code.

Number of programming languages known and familiarity with certain programming concepts also predicted performance. As in the previous experiments, experiential factors were better predictors for those participants with three or fewer years experience programming in Fortran. [Abstract]

[Sites 78]

Richard Sites.

*Programming Tools: Statement Counts and Procedure Timings.**SIGPlan Notices 13(12):98-101*, December, 1978.

It is argued that execution time statement counts and procedure timings are needed in even the first implementation of a high level language. Counts are useful in debugging, algorithm analysis and reliability (for example, which statements have never been executed in testing runs). It is important that the procedure times reflect real time, rather than CPU time, since I/O overhead may present problems which can totally swamp any possible improvements in the CPU-bound part of the code. An example of the usefulness of these techniques is illustrated in the CRAY-1 Pascal compiler. 'It would perhaps be instructive to direct such programmers to take a routine low on the list of percentage of total time, and re-write it to be ten times slower, but 20% smaller and 100% reliable!' [RTS]

- [Slavinski 75] Richard T. Slavinski.
Static Fortran Analyzer.
 Technical Report RADC-TR-75-275, Rome Air Development Center, November, 1975.
 The National Bureau of Standards (NBS) Static FORTRAN Analyzer (SFA), which samples FORTRAN programs and collects statistics on the utilization of predetermined FORTRAN syntactic constructs, was adapted to operate under the FORTRAN-Y compiler of RADC's HIS-635 GCOS operating system. The conversion process and subsequent analysis of 258 sample programs, consisting of approximately 22,000 lines of source code, are provided. The statistical results of this effort may directly support the activities of FORTRAN language study and standardization efforts which address language and compiler design, optimization, and subsetting. [Abstract]
- [Smith 79a] Charles P. Smith.
Practical Applications of Software Science.
 Technical Report TR 03.067, IBM, June, 1979.
 Programming Development at IBM's Santa Teresa Laboratory has been investigating the elements of software science as defined by Maurice H. Halstead. A set of modules have been counted from a large IBM data base program product and the resulting analysis is presented in this report. Program length, vocabulary, volume, difficulty and language level are also discussed as is the possibility of defect prediction in existing code. This paper also discusses some of our problems and concerns. [Abstract]
- [Smith 79b] Connie Smith and J. C. Browne.
Modeling Software Systems for Performance Predictions.
 Technical Report, The University of Texas at Austin, May, 1979.
- [Smith 80] Charles P. Smith.
A Software Science Analysis of IBM Programming Products.
 Technical Report TR-3925, IBM, January, 1980.
 Programming Development at IBM's Santa Teresa Laboratory has been investigating the elements of software science as defined by Maurice H. Halstead. This report summarizes the findings after several large products have been counted. Program length, vocabulary, volume, difficulty, and language level are discussed. [Abstract]
- [Spirn 77] Jeffrey R. Spirn.
Program Behavior: Models and Measurements.
 Elsevier Scientific Publishing Company, 1977.
- [Storey 77] Tony Storey and Stephen Todd.
Performance Analysis of Large Systems.
Software--Practice and Experience 7:323-369, 1977.
 A hybrid analytic and experimental approach to the analysis of large systems is described. The approach is iterative under the assumption that a correct analysis will not be made the first time. The process is 1) make an analysis of the system in terms of basic components; 2) create an estimation model; 3) calculate the cost of the components; 4) create a quantitative estimation model; 5) verify the model experimentally; 6) if verification fails, reiterate; 7) analyze the potential modifications to make an estimation model of the modified system; and 8) evaluate the modifications. [RTS]

- [Svobodova 76] Liba Svobodova.
Computer Performance Measurement and Evaluation Methods: Analysis and Applications.
 Elsevier Scientific Publishing Company, 1976.
- [Thayer 76] T. A. Thayer, et al.
Software Reliability Study.
 Technical Report RADC-TR-76-238, Rome Air Development Center, August, 1976.
 A study of software errors is presented. Techniques for categorizing errors according to type, identifying their source, and detecting them are discussed. Various techniques used in analyzing empirical error data collected from four large software systems are discussed and results of analysis are presented. Use of results to indicate improvements in the error prevention and detection processes through use of tools and techniques is also discussed. [Abstract]
- [Trivedi 79] Kishor S. Trivedi and Robert A. Wagner.
 A Decision Model for Closed Queuing Networks.
IEEE Transactions on Software Engineering SE-5(4):328-332, July, 1979.
 This paper considers a computer configuration design problem. The computer system is modeled by a closed queuing network. The system throughput is the objective function to be maximized and the speed of the devices are the decision variables. A rich class of nonlinear cost functions is considered. It is shown that any local optimum of the optimization problem is also a global optimum. It is also shown that the cost constraint is active and that the method of Lagrange multipliers can be used to solve the problem efficiently. [Abstract]
- [Tuggle 78] Francis D. Tuggle.
 Theory Content and Explanatory Power for Simulation Models.
Behav. Sci. 23(4):271-290, July, 1978.
 A concise methodology for the efficient modeling of behavioral phenomena in living systems at the levels of organisms, groups, organizations, societies, and supranational systems is proposed. The methodology helps resolve questions such as: When a model is to be made more complex? What variables are useful ones to add? When should the model development process cease? The methodology is founded upon 1) the precise identification and delineation of the set of behavioral phenomena to be explained, and 2) the "size" of alternative explanatory models. From these data, measures of explanatory power (relative amount of the phenomena explained) and explanatory yield (the average amount of explained per unit of theory content) may be derived. Explanatory power is shown to be an increasing function of theory content, and explanatory yield is shown to be a decreasing function of theory content. The methodology is illustrated in detail in the context of six successive simulation models of the cognitive behaviors of a subject solving a job shop scheduling task. The success of the final model in providing a complete description of one large class of the subject's behavior corroborates the usefulness of the methodology. [Abstract]
 The above abstract is presented with the published manuscript. Little more need be said; the abstract is quite complete. For individuals interested in modeling behavioral phenomena, or perhaps having an interest in general systems, the paper may be of interest. [CR34458]
- [Van der Knijff 78] D. J. J. Van der Knijff.
 Software Physics and Program Analysis.
Australian Computer Journal 10(3):82-86, August, 1978.
 Software physics is a term used to describe the analysis of programs to extract software engineering measures from particular general properties of the programs. It may be used to compare programs and languages, and to improve estimation procedures in the software industry. This paper introduces the reader to the terms used in software physics and its application to some problems. A selection of recent empirical analyses are presented to enable the reader to make comparisons with other methods.

[CR33992]

[Waite 73]

W.M. Waite.

A Sampling Monitor for Applications Programs.

Software--Practice and Experience 3:75-79, 1973.

A set of monitoring conventions are specified for sampling. If the operating system does not allow interrupt handling by the user, it is necessary to put some of the routines in the monitor. A set of interface conventions for such a facility is described. [RTS]

[Walston 77]

C. E. Walston and C. P. Felix.

A Method of Programming Measurement and Estimation.

IBM Systems Journal 16(1), 1977.

Improvements in programming technology have paralleled improvements in computing system architecture and materials. Along with increasing knowledge of the system and program development processes, there has been some notable research into programming project productivity estimation. Also presented are preliminary results of research into methods of measuring and estimating programming project productivity estimation. Also presented are preliminary results of research into methods of measuring and estimating programming project duration, staff size, and computer cost. [VRB]

[Walters 78]

Gene F. Walters and James A. McCall.

The Development of Metrics for Software Reliability and Maintainability.

In *Proceedings of the 1978 Reliability and Maintainability Symposium*, 1978.

[Waters 79]

Richard C. Waters.

A Method for Analyzing Loop Programs.

IEEE Transactions on Software Engineering SE-5(3):237-247, May, 1979.

This paper presents a method for automatically analyzing loops, and discusses why it is a useful way to look at loops. The method is based on the idea that there are four basic ways in which the logical structure of a loop is built up. An experiment is presented which shows that this accounts for the structure of a large class of loops. The paper discusses how the method can be used to automatically analyze the structure of a loop, and how the resulting analysis can be used to guide a proof of correctness for the loop. An automatic system is described which performs this type of analysis. The paper discusses the relationship between the structure building methods presented and designed to assist a person who is writing a program. The intent is that the system will cooperate with a programmer throughout all phases of work on a program and be able to communicate with the programmer about it. [Abstract]

[Weiss 79]

David M. Weiss.

Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility.

The Journal of Systems and Software(1), 1979.

In software engineering, it is easy to propose techniques for improving software development but difficult to test the claims made for such techniques. This paper suggests an error analysis technique for use in gathering data concerning the effectiveness of different software development methodologies. The principal features of the error analysis technique described are the formulation of questions of interest and a data classification scheme before data collection begins, and interviews of system developers concomitant with the development process to verify the accuracy of the data. The data obtained by using this technique during the development of a medium-size software development project is presented. This project was known as the Architecture Research Facility (ARF) and took about 10 months and 192 man-weeks of effort to develop. The ARF designers used the information hiding principle to modularize the system, and interface specifications and high-level language coding specifications to express the design. Several error detection aids were designed into the system to help detect run-time errors. In addition, quality control rules were established that required review of specifications before coding, and review of code after compilation but prior to

testing. A total of 143 errors was reported. Analysis of these errors showed that there were few problems caused by intermodule interfaces, that error corrections rarely required knowledge of more than one module, that most errors took less than a few hours to fix, and that the error detection aids detected more than half of the errors that were potentially detectable by them. [VRB]

- [Weissman 73] Lawrence M. Weissman.
Psychological Complexity of Computer Programs: An Initial Experiment.
 Technical Report CSRG-26, University of Toronto, July, 1973.
 In order to reduce the complexity of programs many ideas and techniques have been expounded. However, no quantitative evidence has been given that the quality of the programs has indeed been improved. We believe that experimental studies should be performed to measure those factors which make programs complex. An initial experiment has been conducted to measure the effects of three such factors, comments, paragraphing, and mnemonic variable names. This report summarizes the results of this experiment. [Abstract]
- [Weissman 74] Laurence M. Weissman.
A Methodology for Studying the Psychological Complexity of Computer Programs.
 Technical Report CSRG-37, University of Toronto, August, 1974.
 There are many reasons for empirically testing hypotheses about the effects of various factors on the psychological complexity of computer programs. (By "psychological complexity" we mean the intrinsic property of programs that affects their understandability and maintainability.) This thesis develops a methodology for such experimentation, and discusses the results of ten experiments involving the following factors: use of comments, control flow paragraphing, choice of variable names, and locality of data references. [Abstract]
- [Wichmann 69] Brian A. Wichmann.
A Comparison of Algol 60 Execution Speeds.
 Technical Report CCU-3, National Physical Laboratory, January, 1969.
- [Wichmann 70] Brian A. Wichmann.
Some Statistics from Algol Programs.
 Technical Report CCU-11, National Physical Laboratory, August, 1970.
 Reports on dynamic analysis of 949 programs and static analysis of 40 programs. [MS]
- [Willman 77] H. E. Willman, Jr., T. A. James, A. A. Beauregard and P. Hilcoff.
Software Systems Reliability: A Raytheon Project History.
 Technical Report RADC-TR-77-188, Rome Air Development Center, June, 1977.
 This report presents results of a project to collect software data from the records of development of a large Department of Defense ground-based system. A description of the subject systems software development process, characteristics, tools, and test methods are presented. Qualitative and quantitative data gathered from configuration management files are included as well as statistical summaries of this data. A detailed description of the data base files is included as well as portions of the actual data base. Recommendations are made for the use of the data as well for the future collection of such data. [Abstract]
- [Wolverton 74] Ray W. Wolverton.
The Cost of Developing Large-Scale Software.
 IEEE Transactions on Computers 23(6), 1974.
 The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgement, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 x 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented

in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five traditional methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, and bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested. [VRB]

[Wong 74]

K. Wong and J.C. Strauss.

Use of a Software Monitor in the Validation of an Analytic Computer System Model.
Software--Practice and Experience 4:255-263, 1974.

A sampling monitor is described which examines OS/360 system tables and control blocks periodically for CPU activity, the priority mapping of certain tasks, I/O queuing activity and I/O activity of the devices on the selector channels. The monitor is a normal task which is loaded into a 20K partition with high priority. The data derived from the monitoring process was then used to validate an analytic model. [RTS]

[Woodfield 79]

Scott N. Woodfield.

An Experiment on Unit Increase in Problem Complexity.

IEEE Transactions on Software Engineering SE-5(2):76-79, March, 1979.

The effect of a variation in problem complexity and how the variation relates to programming complexity is predicted and measured. An experiment was conducted in which eighteen graduate students programmed two variations of the same small algorithm where the problem complexity varied by 25 percent. Eight measurable program characteristics are compared with predicted values obtained using only two known parameters. The agreement between observed and predicted values is very good. Both predicted and observed measurements indicate that the 25 percent increase in problem complexity results in a 100 percent increase in programming complexity. [Abstract]

[Woodward 79]

Martin R. Woodward, Michael A. Hennell and David Hedley.

A Measure of Control Flow Complexity in Program Text.

IEEE Transactions on Software Engineering SE-5(1):45-50, January, 1979.

This paper discusses the need for measures of complexity and unstructuredness of programs. A simple language independent concept is put forward as a measure of control flow complexity in program text and is then developed for use as a measure of unstructuredness. The proposed metric is compared with other metrics, the most notable of which is the cyclomatic complexity measure. Some experience with automatic tools for obtaining these metrics is reported. [Abstract]

[Wortman 76]

D. B. Wortman.

A Study of High-Resolution Timing.

IEEE Transactions on Software Engineering SE-2:135-137, June, 1976.

This article describes an experimental comparison of timing information provided by a large multiprogramming system (OS/370 MVT) with timing information derived directly from a high resolution hardware clock. The hardware clock was found to be a superior source of timing information. [Abstract]

[Wulf ??] W. Wulf, P. Feiler, J. Zinnikas, R. Brender.
A Quantitative Technique For Comparing the Quality of Language Implementations.
In preparation.

[Yuval 75] G. Yuval.
Gathering Run-Time Statistics Without Black Magic.
Software--Practice and Experience:105-108, 1975.

The Pascal/6000 compiler was modified to add a 'turnstile' program, a piece of code that will count how often it has been passed through, to the prologue of each routine. A postprocessor is used to construct a profile by searching through memory looking for turnstiles. The CDC 6000 smallest turnstile is 1 word (containing a subroutine call and a counter); the fastest is 105 bits long (a word is 60 bits long) and takes 1.2 to 1.5 microseconds. [RTS]

[Zweben 79] Stuart H. Zweben and Maurice H. Halstead.
The Frequency Distribution of Operators in PL/1 Programs.
IEEE Transactions on Software Engineering SE-5(2):91-95, March, 1979.

During the past few years, several investigators have noted definite patterns in the distribution of operators in computer programs. Their proposed models have provided explanations for other observed software phenomena and have suggested possible relationships between programming languages and natural languages. However, these models contain notable deficiencies.

This study concentrates on a set of production programs written in PL/1. Using some basic relationships from software science, and a previously published algorithm generation technique, a model for computing operator frequencies is constructed which is based only on the number of distinct operators in the program and the total number of operator occurrences. The model provides a considerable statistical improvement over existing models for the PL/1 programs studied. [Abstract]